宝典丛书 300万

Python 3

宝典

基于最新的Python 3.x版本进行讲解，与2.x有区别的地方附有相关说明，适合2.x和3.x的读者。

涵盖用Python操作数据库、进行Web开发、网络编程、科学计算、多线程编程等高端领域。

书中提供三大案例，分别涉及用Python进行Windows优化、大数据处理和游戏开发。

杨佩璐 宋 强 等编著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Python宝典

















上架建议：计算机>Python

ISBN 978-7-121-22562-8

9 787121 225628 >

策划编辑：张月萍
责任编辑：徐津平
封面设计：秦　靖

定价：79.80元

· 014040387

TP311.56
1163

宝典丛书

Python 宝典

杨佩璐　宋强　等编著

北京航空航天大学
藏书
图书馆

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

TP311.56
1163

北航 C1724451

内 容 简 介

Python 是目前流行的脚本语言之一。本书由浅入深、循序渐进地为读者讲解了如何使用 Python 进行编程开发。全书内容共分三篇，分为入门篇、高级篇和案例篇。入门篇包括 Python 的认识和安装、开发工具简介、Python 基本语法、数据结构与算法、多媒体编程、系统应用、图像处理和 GUI 编程等内容。高级篇包括用 Python 操作数据库、进行 Web 开发、网络编程、科学计算、多线程编程等内容。案例篇选择了 3 个案例演示了 Python 在 Windows 系统优化、大数据处理和游戏开发方面的应用。

本书针对 Python 的常用扩展模块给出了详细的语法介绍，并且给出了典型案例，通过对本书的学习，读者能够很快地使用 Python 进行编程开发。

本书适合 Python 初学者、程序设计人员、编程爱好者、本科及大专院校学生，以及需要进行对科学的计算的工程人员阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Python 宝典 / 杨佩璐等编著. —北京：电子工业出版社，2014.5
（宝典丛书）
ISBN 978-7-121-22562-8

Ⅰ. ①P… Ⅱ. ①杨… Ⅲ. ①软件工具—程序设计 Ⅳ. ①TP311.56

中国版本图书馆 CIP 数据核字（2014）第 047661 号

策划编辑：张月萍
责任编辑：徐津平
印　　刷：三河市鑫金马印装有限公司
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787×1092　1/16　　印张：31.5　　字数：850千字
印　　次：2014年5月第1次印刷
定　　价：79.80元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888

前　言

Python 是一种功能强大的脚本语言。使用 Python 可以完成从文本处理到创建复杂的三维图形等各种工作。在企业级应用中，由于 Python 具有简洁的语法和丰富的扩展模块，因此，使用 Python 可以大幅缩短开发周期，节约成本。

另外，Jython 还可以在 Java 中使用 Python，通过 Python 的灵活性来提高 Java 在企业级应用的效率。在 Web 方面，有很多基于 Python 的流行 Web 框架，如 Zope/Plone、Django、TurboGear 等。通过这些 Web 框架，程序员可以使用 Python 快速构建安全、功能强大的网站。

在数值计算与工程应用方面，Python 与传统的 C 和 Fortran 相比，更加灵活、简洁，并且可以十分方便地创建 GUI 界面。通过使用 SciPy 模块和 Matplotlib 绘图库可以进行数值计算，实现工程数据的可视化。

如何学习本书

本书内容共分三篇 26 章，分别为入门篇、高级篇和案例篇。

入门篇为第 1 章至第 15 章，从初识 Python 开始，由浅入深地介绍了 Python 的安装、开发工具的使用、Python 数据类型与基本语句、可复用的函数与模块、数据结构与算法、面向对象的 Python、异常处理与程序调试、Python 多媒体编程、使用 PIL 处理图片、系统编程、使用 Python 的 GUI 编程等内容。其中，第 11 章至第 15 章专门对 Python 的几种 GUI 编程工具进行了讲解，读者可以比较各种 GUI 编程工具，根据自己的兴趣及以前所学的知识，选择适合自己使用的 GUI 编程工具。

高级篇为第 16 章至第 23 章，主要介绍了用 Python 操作数据库、进行 Web 开发、网络编程、科学计算、多线程编程等内容。其中，第 22 章介绍了 Python 的扩展和嵌入，需要读者有 C/C++ 的相关背景，如果读者不会使用 C/C++进行编程，可以跳过该章。在大部分开发场景中，不掌握这部分知识也不影响用 Python 编程。

案例篇为第 24 章至第 26 章，每章介绍一个案例，包括用 Python 优化 Windows、用 Python 处理大数据和用 Python 开发《植物大战僵尸》游戏。通过对这 3 个案例的学习，可进一步巩固读者前面所学的知识。

本书以 Python 3.x 为基础进行讲解，并在与 Python 2.x 有区别的地方加上了相关介绍，使 Python 2.x 和 Python 3.x 的读者都能使用本书。

本书特色

- 内容全面，对 Python 各方面的知识都做了系统详尽的讲解。

- 结构清晰，全书整体结构上遵循从易到难、由浅入深的顺序。
- 内容新颖，结合当前最新的 Python 3.x 版本进行讲解。
- 实用性强，本书在各章节中都有大量程序示例，并在最后一篇详细讲解了 3 个不同方向的开发实例。
- 实例丰富，对于每一个知识点，书中都通过相应的示例进行讲解。

适合的读者

- Python 初学者
- 程序设计人员
- 编程爱好者
- 本科及大专院校学生
- 需要进行科学计算的工程人员

本书由杨佩璐（山东中医药大学）、宋强（安阳工学院）共同编写，其中杨佩璐编写了本书的第 1~13 章，宋强编写了本书的第 14~20 章，同时参与编写的还有徐新其、张俊华、赵桂芹、张增强、刘桂珍、李杰、曾智海、代得新、邵星星、过建军、王正江、朱林鑫、张伟杰、田亮亮，在此一并表示感谢。

在本书的编写过程中，编者竭尽全力，不敢有丝毫疏忽，并对所有程序都进行了上机调试，但由于 Python 发展非常迅速，仍会有不同版本的差别，因此需要读者多加注意。另外，书中难免有不足之处，望广大读者批评指正。

编者

2014 年 1 月

目　录

第 1 部分　入门篇

第 1 章　初识 Python 2

1.1　Python 是什么 2
1.2　Python 有什么优点 3
1.3　其他程序设计语言中的 Python 4
1.4　快速搭建 Python 开发环境 5
1.4.1　哪些系统中可使用 Python 5
1.4.2　Python 的下载和安装 6
1.4.3　用 VS2008 编译 Python 源码 8
1.4.4　Python 开发工具：Vim 9
1.4.5　Python 开发工具：Emacs 13
1.4.6　Python 开发工具：PythonWin 16
1.4.7　其他的 Python 开发工具 17
1.5　第一个 Python 程序 19
1.5.1　从“Hello，Python!”开始 19
1.5.2　Python 的交互解释器 20
1.6　本章小结 21

第 2 章　Python 起步必备 22

2.1　Python 代码的组织形式 22
2.1.1　用缩进来分层 22
2.1.2　两种代码注释的方式 23
2.1.3　Python 语句的断行 23
2.2　Python 的基本输入输出函数 25
2.2.1　接收输入的 input 函数 25
2.2.2　输出内容的 print 函数 26
2.3　Python 对中文的支持 27
2.3.1　Python 3 之前版本如何使用中文 27
2.3.2　更全面的中文支持 29
2.4　简单实用的 Python 计算器 29
2.4.1　直接进行算术运算 30
2.4.2　math 模块提供丰富的数学函数 30
2.4.3　Python 对大整数的支持 31
2.5　本章小结 32

第 3 章　Python 数据类型与基本语句 33

3.1　Python 数据类型：数字 33
3.1.1　整型和浮点型 33
3.1.2　运算符 34
3.2　Python 数据类型：字符串 36
3.2.1　Python 中的字符串 36
3.2.2　字符串中的转义字符 36
3.2.3　操作字符串 37
3.2.4　字符串的索引和分片 39
3.2.5　格式化字符串 40
3.2.6　字符串、数字类型的转换 40
3.2.7　原始字符串（Raw String） 41
3.3　Python 数据类型：列表和元组 42
3.3.1　创建和操作列表 42
3.3.2　创建和操作元组 43
3.4　Python 数据类型：字典 43
3.5　Python 数据类型：文件 44
3.6　Python 的流程控制语句 46
3.6.1　分支结构：if 语句 46
3.6.2　循环结构：for 语句 48
3.6.3　循环结构：while 语句 50
3.7　本章小结 51

第 4 章　可复用的函数与模块 52

4.1　Python 自定义函数 52
4.1.1　函数声明 52
4.1.2　函数调用 53
4.2　参数让函数更有价值 54
4.2.1　有默认值的参数 54
4.2.2　参数的传递方式 55
4.2.3　如何传递任意数量的参数 56
4.2.4　用参数返回计算结果 57
4.3　变量的作用域 57
4.4　最简单的函数：用 lambda 声明函数 58
4.5　可重用结构：Python 模块 59

4.5.1 Python 模块的基本用法.....59
4.5.2 Python 在哪里查找模块.....61
4.5.3 是否需要编译模块.....62
4.5.4 模块也可独立运行.....63
4.5.5 如何查看模块提供的函数名.....64
4.6 用包来管理多个模块.....65
4.7 本章小结.....66

第 5 章 数据结构与算法.....67

5.1 表、栈和队列.....67
5.1.1 表.....67
5.1.2 栈.....68
5.1.3 队列.....70
5.2 树和图.....72
5.2.1 树.....72
5.2.2 二叉树.....73
5.2.3 图.....76
5.3 查找与排序.....78
5.3.1 查找.....78
5.3.2 排序.....79
5.4 本章小结.....82

第 6 章 面向对象的 Python.....83

6.1 面向对象编程概述.....83
6.1.1 Python 中的面向对象思想.....83
6.1.2 类和对象.....84
6.2 在 Python 中定义和使用类.....84
6.2.1 类的定义.....85
6.2.2 类的使用.....86
6.3 类的属性和方法.....87
6.3.1 类的属性.....87
6.3.2 类的方法.....88
6.4 类的继承.....91
6.4.1 使用继承.....91
6.4.2 Python 的多重继承.....92
6.5 在类中重载方法和运算符.....94
6.5.1 方法重载.....94
6.5.2 运算符重载.....95
6.6 在模块中定义类.....97
6.7 本章小结.....98

第 7 章 异常处理与程序调试.....99

7.1 异常的处理.....99
7.1.1 用 try 语句捕获异常.....99
7.1.2 常见异常的处理.....101
7.1.3 多重异常的捕获.....102
7.2 用代码抛出异常.....103
7.2.1 用 raise 抛出异常.....103
7.2.2 assert——简化的 raise 语句.....104
7.2.3 自定义异常类.....105
7.3 使用 pdb 调试 Python 脚本.....106
7.3.1 运行语句.....106
7.3.2 运行表达式.....107
7.3.3 运行函数.....107
7.3.4 设置硬断点.....108
7.3.5 pdb 调试命令.....109
7.4 在 PythonWin 中调试程序.....111
7.5 本章小结.....113

第 8 章 Python 多媒体编程.....114

8.1 使用 PyOpenGL 绘制三维图形.....114
8.1.1 安装 PyOpenGL.....114
8.1.2 使用 PyOpenGL 创建窗口.....115
8.1.3 绘制文字.....116
8.1.4 绘制二维图形.....118
8.1.5 绘制三维图形.....120
8.1.6 纹理映射.....122
8.2 播放音频文件.....125
8.2.1 使用 DirectSound.....125
8.2.2 使用 WMPlayer.OCX.....126
8.3 PyGame.....128
8.3.1 安装 PyGame.....128
8.3.2 使用 PyGame 编写简单的游戏.....129
8.4 本章小结.....132

第 9 章 使用 PIL 处理图片.....133

9.1 PIL 概述.....133
9.1.1 安装 PIL.....133
9.1.2 PIL 简介.....135
9.2 使用 PIL 处理图片.....137
9.2.1 转换图片格式.....137
9.2.2 生成缩略图.....139
9.2.3 为图片添加 Logo.....142
9.3 本章小结.....147

第 10 章 系统编程 148

10.1 访问 Windows 注册表 148
10.1.1 注册表概述 148
10.1.2 使用 Python 操作注册表 149
10.1.3 查看系统启动项 152
10.1.4 修改 IE 153
10.2 文件和目录 156
10.2.1 文件目录常用函数 156
10.2.2 批量重命名 158
10.2.3 代码框架生成器 159
10.3 生成可执行文件 160
10.3.1 安装 py2exe 161
10.3.2 使用 py2exe 生成可执行文件 161
10.3.3 使用 cx_freeze 生成可执行文件 163
10.4 运行其他程序 164
10.4.1 使用 os.system()函数运行其他程序 164
10.4.2 使用 ShellExecute 函数运行其他程序 165
10.4.3 使用 CreateProcess 函数运行其他程序 166
10.4.4 使用 ctypes 调用 kernel32.dll 中的函数 167
10.5 本章小结 168

第 11 章 使用 PythonWin 编写 GUI 169

11.1 Windows GUI 编程概述 169
11.1.1 使用 Windows API 创建窗口 169
11.1.2 使用 MFC 创建窗口 172
11.2 创建对话框 172
11.2.1 创建对话框 173
11.2.2 向对话框添加控件 174
11.2.3 使用 DLL 文件中的资源 176
11.2.4 处理按钮消息 177
11.3 创建菜单 179
11.3.1 创建菜单 179
11.3.2 使用 DLL 中的菜单 182
11.3.3 处理菜单消息 184
11.4 本章小结 185

第 12 章 使用 tkinter 编写 GUI 186

12.1 tkinter 概述 186
12.1.1 创建简单的窗口 186
12.1.2 向窗口中添加组件 187
12.2 使用组件 188
12.2.1 组件分类 188
12.2.2 组件布局 188
12.2.3 使用按钮 189
12.2.4 使用文本框 190
12.2.5 使用标签 192
12.2.6 使用菜单 193
12.2.7 使用单选框和复选框 195
12.2.8 绘制图形 197
12.3 事件处理 199
12.3.1 事件表示 199
12.3.2 响应事件 201
12.4 创建对话框 204
12.4.1 使用标准对话框 204
12.4.2 创建自定义对话框 208
12.5 本章小结 210

第 13 章 使用 wxPython 编写 GUI 211

13.1 wxPython 概述 211
13.1.1 安装 wxPython 211
13.1.2 创建窗口 212
13.2 组件 214
13.2.1 面板 214
13.2.2 按钮 215
13.2.3 标签 217
13.2.4 文本框 218
13.2.5 单选框和复选框 221
13.2.6 使用 sizer 布置组件 222
13.3 对话框 224
13.3.1 消息框和标准对话框 224
13.3.2 创建自定义对话框 226
13.4 菜单 227
13.4.1 创建菜单 228
13.4.2 绑定菜单事件 230
13.5 一个简单的文本编辑器 231
13.6 本章小结 234

第 14 章 使用 PyGTK 编写 GUI 235

14.1 PyGTK 概述 235
14.1.1 PyGTK 安装 235
14.1.2 创建窗口 236

14.2 组件 238
14.2.1 标签 238
14.2.2 按钮 241
14.2.3 容器组件 243
14.2.4 文本框 246
14.2.5 单选框和复选框 249
14.3 消息框和对话框 250
14.3.1 消息框 250
14.3.2 标准对话框 252
14.3.3 自定义对话框 254
14.4 使用菜单 256
14.4.1 创建菜单 256
14.4.2 菜单事件 259
14.5 资源文件 260
14.5.1 使用 Glade 创建资源文件 261
14.5.2 使用资源文件 263
14.6 本章小结 264

第 15 章 使用 PyQT 编写 GUI 265

15.1 PyQt 概述 265
15.1.1 PyQt 的安装 265
15.1.2 使用 PyQt 创建窗口 266
15.2 组件 267
15.2.1 标签 267
15.2.2 布局组件和空白项 268
15.2.3 按钮 270
15.2.4 文本框 272
15.2.5 单选框和复选框 275
15.2.6 菜单 276
15.3 创建对话框 278
15.3.1 消息框和标准对话框 279
15.3.2 自定义对话框 283
15.4 使用资源 285
15.4.1 使用 Qt Designer 创建资源文件 285
15.4.2 使用资源文件 287
15.5 本章小结 288

第 2 部分 高级篇

第 16 章 Python 与数据库 290

16.1 连接 Access 数据库 290
16.1.1 使用 ODBC 连接 Access 数据库 290
16.1.2 使用 DAO 连接 Access 数据库 294
16.1.3 使用 ADO 连接 Access 数据库 295
16.2 使用 MySQL 数据库 296
16.2.1 安装 MySQL 297
16.2.2 连接到 MySQL 299
16.3 嵌入式数据库 SQLite 301
16.4 本章小结 302

第 17 章 Python Web 应用 303

17.1 开源 Web 应用服务器 Zope 303
17.1.1 安装 Zope 303
17.1.2 使用 Zope 管理界面 305
17.1.3 创建模板 308
17.1.4 添加 Python 脚本 310
17.2 使用 Plone 内容管理系统 312
17.2.1 安装 Plone 312
17.2.2 安装 Plone 插件 314
17.3 在 Microsoft IIS 中使用 Python 316
17.3.1 安装 Microsoft IIS 317
17.3.2 在 ASP 中使用 Python 脚本 319
17.3.3 一个简单的例子 321
17.4 在 Apache 中使用 Python 325
17.4.1 安装配置 Apache 325
17.4.2 安装 mod_python 327
17.4.3 使用 Python Sever Pages 创建留言板 328
17.5 本章小结 331

第 18 章 Python 网络编程 332

18.1 使用 socket 模块 332
18.1.1 网络编程概述 332
18.1.2 使用 socket 模块建立网络通信 333
18.1.3 在局域网中传输文件 338
18.2 使用 urllib、httplib 和 ftplib 341
18.2.1 使用 Python 访问网站 341
18.2.2 访问 FTP 345
18.3 使用 poplib 和 smtplib 模块收发邮件 350
18.3.1 检查 E-mail 350

18.3.2 发送 E-mail.....353
18.4 本章小结.....357

第 19 章 处理 HTML 与 XML.....358

19.1 处理 HTML.....358
19.1.1 HTMLParser 类简介.....358
19.1.2 获取页面图片地址.....359
19.1.3 查看天气预报.....361
19.2 处理 XML.....366
19.2.1 XML 基础.....367
19.2.2 文档类型定义.....368
19.2.3 命名空间.....370
19.3 使用 Python 处理 XML.....370
19.3.1 使用 xml.parsers.expat 处理 XML.....371
19.3.2 使用 xml.sax 处理 XML.....373
19.3.3 使用 xml.dom 处理 XML.....374
19.4 简单的 RSS 阅读器.....375
19.5 本章小结.....378

第 20 章 功能强大的正则表达式.....379

20.1 正则表达式概述.....379
20.1.1 正则表达式的基本元字符.....379
20.1.2 常用正则表达式分析.....380
20.2 支持正则表达式的 re 模块.....381
20.2.1 用 match 函数进行搜索.....381
20.2.2 用 sub 函数进行内容替换.....382
20.2.3 用 split 函数分割字符串.....383
20.3 编译生成正则表达式对象.....383
20.3.1 以“\”开头的元字符.....383
20.3.2 用 compile 函数编译正则表达式.....385
20.3.3 在正则表达式中使用原始字符串.....385
20.4 用正则表达式对象提速.....386
20.4.1 使用 match 方法匹配和搜索.....386
20.4.2 使用 sub 方法替换内容.....387
20.4.3 使用 split 方法分割字符串.....388
20.5 正则表达式中的分组.....389
20.5.1 分组的概述.....389
20.5.2 分组的扩展语法.....390
20.6 匹配和搜索的结果对象：Match 对象.....391
20.6.1 使用 Match 对象处理组.....391
20.6.2 使用 Match 对象处理索引.....392
20.7 使用正则表达式处理文件.....393
20.8 本章小结.....395

第 21 章 科学计算.....396

21.1 NumPy 和 SciPy 简介.....396
21.1.1 安装 NumPy 和 SciPy.....396
21.1.2 NumPy 简介.....398
21.1.3 SciPy 简介.....399
21.2 矩阵运算和解线性方程组.....400
21.2.1 矩阵运算.....400
21.2.2 解线性方程组.....402
21.3 使用 Matplotlib 绘制函数图形.....403
21.3.1 安装 Matplotlib.....403
21.3.2 使用 Matplotlib 绘制图形.....405
21.4 本章小结.....407

第 22 章 Python 扩展和嵌入.....408

22.1 用 C/C++扩展 Python.....408
22.1.1 VS2008 编译环境的设置.....408
22.1.2 Python 扩展程序的结构.....414
22.1.3 在 Python 扩展中使用 MFC.....416
22.2 在 C/C++中嵌入 Python.....420
22.2.1 高层次的嵌入 Python.....420
22.2.2 较低层次嵌入 Python.....421
22.2.3 在 C 中嵌入 Python 实例.....426
22.3 通过 SWIG 编写 Python 扩展.....428
22.3.1 在 VS 中使用 SWIG.....428
22.3.2 SWIG 接口文件的语法简介.....431
22.4 Boost.Python 使程序更简单.....433
22.4.1 下载编译 Boost.Python.....433
22.4.2 使用 Boost.Python 简化扩展和嵌入.....435
22.4.3 使用 Pyste 生成代码.....439
22.5 本章小结.....440

第 23 章 多线程编程.....441

23.1 线程基础.....441
23.1.1 创建线程.....441
23.1.2 Thread 对象中的方法.....442
23.2 线程同步.....445
23.2.1 简单的线程同步.....445
23.2.2 使用条件变量保持线程同步.....447

23.2.3 使用队列让线程同步 448
23.3 线程间通信.....449
23.3.1 Event 对象的方法.....449
23.3.2 使用 Event 对象实现线程间通信 450
23.4 微线程——Stackless Python.....450
23.4.1 Stackless Python 概述 451
23.4.2 使用微线程 453
23.5 本章小结.....454

第 3 部分 案例篇

第 24 章 案例 1：用 Python 优化 Windows 456

24.1 案例概述.....456
24.2 创建图形化界面.....457
24.2.1 编写脚本创建 GUI 457
24.2.2 响应菜单事件 459
24.3 清理垃圾文件.....461
24.3.1 遍历目录 462
24.3.2 扫描垃圾文件 463
24.3.3 使用多线程 464
24.3.4 扫描所有磁盘 465
24.3.5 删除垃圾文件 467
24.4 搜索文件.....469
24.4.1 搜索大文件 469
24.4.2 按名称搜索文件 471
24.5 本章小结.....472

第 25 章 案例 2：用 Python 玩转大数据.....473

25.1 案例概述.....473
25.1.1 了解大数据处理方式 473
25.1.2 处理日志文件 474
25.1.3 案例目标.....475
25.2 日志文件的分割.....476
25.3 编写 Map 函数处理小文件.....477
25.4 编写 Reduce 函数.....479
25.5 本章小结.....480

第 26 章 案例 3：植物大战僵尸 481

26.1 案例概述.....481
26.1.1 游戏效果.....481
26.1.2 游戏规划设计 482
26.2 收集资源.....483
26.2.1 收集图片素材 483
26.2.2 收集声效素材 484
26.3 编写初始脚本.....484
26.3.1 定义游戏初始环境 484
26.3.2 导入游戏素材 486
26.4 编写游戏核心脚本 488
26.4.1 编写游戏循环脚本 488
26.4.2 处理事件——响应玩家的操作.....489
26.4.3 添加角色到游戏 490
26.4.4 更新角色状态 491
26.4.5 重绘画面.....493
26.4.6 判断角色交战状态 493
26.4.7 判断胜负状态 494
26.5 本章小结.....494

Part

第 1 部分　入门篇

第 1 章　初识 Python
第 2 章　Python 起步必备
第 3 章　Python 数据类型与基本语句
第 4 章　可复用的函数与模块
第 5 章　数据结构与算法
第 6 章　面向对象的 Python
第 7 章　异常处理与程序调试
第 8 章　Python 多媒体编程
第 9 章　使用 PIL 处理图片
第 10 章　系统编程
第 11 章　使用 PythonWin 编写 GUI
第 12 章　使用 tkinter 编写 GUI
第 13 章　使用 wxPython 编写 GUI
第 14 章　使用 PyGTK 编写 GUI
第 15 章　使用 PyQT 编写 GUI

1

第 1 章 初识 Python

本章包括

- 什么是 Python
- 其他程序设计语言中的 Python
- 用 VS2008 编译 Python 源码
- 第一个 Python 程序
- Python 的优点
- Python 的下载和安装
- Python 开发工具：Vim、Emacs 和 PythonWin

Python 是一种面向对象的、直译式计算机程序设计语言，也是一种功能强大且完善的通用型语言，已经具有二十多年的发展历史，成熟且稳定。Python 是免费的语言，具有面向对象的特性，可以运行在多种操作系统之上。Python 具有清晰的结构、简洁的语法以及强大的功能。Python 可以完成从文本处理到网络通信等各种工作。Python 自身已经提供了大量的模块来实现各种功能，除此之外，还可以使用 C/C++来扩展 Python，甚至还可以将 Python 嵌入到其他语言中。

1.1 Python 是什么

Python 是目前流行脚本语言之一，它是由 Guido van Rossum 创建的。Python 语言主要受到教学语言 ABC 和 Modula-3 的影响。Python 被设计得简洁、优美却又不失脚本的灵活性和拥有强大的功能。Python 的主要特点可以用图 1-1 来表示。

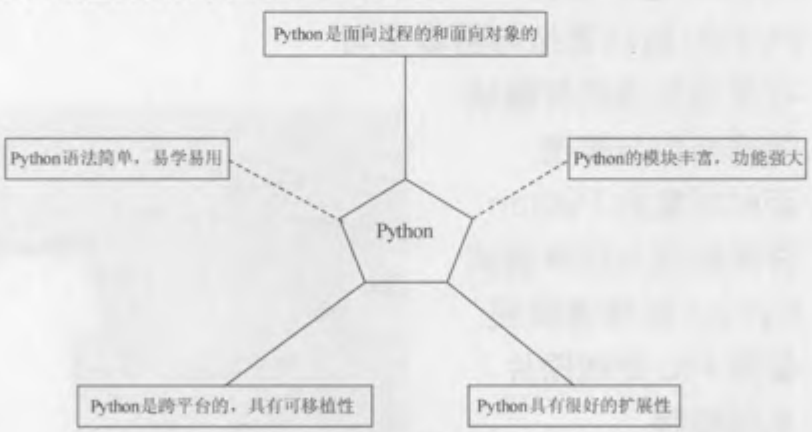


图 1-1 Python 的主要特点

Python 最大的特点是其独特而又简洁的语法。在 Python 源代码中，通过代码在行中不同的缩进量来表示代码所属的语句块，从而不需要使用类似 C/C++中的花括号 "{}"。这个特点对于学过 C/C++，或长期使用 C/C++的用户来说，可能会有一点不适应。但是，Python 强制性的缩进可使代码看起来更加清晰，并且使用缩进在一定程度上也减少了代码输入量（因为省去了一对花括号），当然，前提是所使用的代码编辑器支持自动缩进。

使用 Python 可以非常快速地进行编程开发。Python 的语法很简单，对于初学者来说，学习并使用 Python 并不困难。甚至仅花费几天时间对 Python 进行短暂的学习，就可以使用 Python 编写出“引以为傲”的非常实用的脚本。

作为脚本语言，Python 非常灵活，使用 Python 可以实现各种各样的功能。通常来说，脚本语言都是解释型的语言，不需要编译过程。虽然 Python 被称之为脚本语言，但 Python 具有编译过程，Python 会将脚本编译成字节码的形式。并且，一般不需要程序员对脚本显式地进行编译，Python 自己会根据需要编译（通常情况下，只有作为模块的脚本才会被编译成字节码的形式）。

Python 遵循 GPL 协议，是源代码开放的软件。用户不仅可以免费使用 Python 来编写脚本，还可以阅读 Python 的源代码，了解 Python 的内部代码。用户甚至还可以参与到 Python 的开发之中，为 Python 的发展做出贡献。

1.2 Python 有什么优点

虽然 Python 不是程序语言唯一的选择，但却是一种不错的选择。为什么使用 Python 作为编程语言？这首先取决于用户。面对简单易学且功能强大的 Python，越来越多的人选择使用它来完成各种各样的任务。

1. Python 是免费的自由软件

Python 遵循 GPL 协议，是自由软件，这是 Python 流行的原因之一。用户使用 Python 进行开发和发布自己编写的程序不需要支付任何费用，不用担心版权问题。即使作为商业用途，Python 也是免费的。开源的自由软件正在成为软件行业的一种发展趋势，现在，很多商业软件公司也开始将自己的产品变为开源的，如 Java。作为开源软件的 Python 将具有更强的生命力。

作为自由软件，最令人鼓舞的就是可以很方便地获取源代码。程序员通过阅读其源代码，发现其中的神奇之处。

当深入地使用 Python 以后，可能会发现 Python 的某些特性，而这些特性并没有详细的说明文档，此时可以阅读 Python 的源代码去详细地了解 Python 的这些特性。

2. Python 是跨平台的

跨平台、良好的可移植性是 C 语言成为经典编程语言的关键，而 Python 正是用可移植的 ANSI C 编写的，这意味着 Python 也具有良好的跨平台特性。也就是说，在 Windows 下编写的 Python 脚本可以轻易地运行在 Linux 下。当然，如果在 Python 脚本中使用了 Windows 的某些特性（如 COM），那就另当别论了。

Python 不仅能在 Windows、Linux 系统中运行，由于它的开源本质，已经被移植在许多平台上，包括 Linux、Windows、FreeBSD、Macintosh、Solaris、OS/2、Amiga、AROS、AS/400、BeOS、OS/390、z/OS、Palm OS、QNX、VMS、Psion、Acorn RISC OS、VxWorks、PlayStation、Sharp Zaurus、Windows CE、PocketPC、Symbian 以及 Google 基于 Linux 开发的 Android 平台。不难看出，Python 不仅能够运行在传统的 Server、PC 系统中，还能够运行在正蓬勃发展的各类移动系统中。

3. Python 功能强大

Python 强大的功能也许才是很多用户支持 Python 的最重要的原因。从字符串处理到复杂的 3D 图形编程，Python 借助扩展模块都可以轻松完成。实际上，Python 的核心模块已经提供了足够强

大的功能，使用 Python 精心设计的内置对象可以完成许多功能强大的操作。

基于强大的功能，Python 可以使用在众多领域，例如：

- 系统编程，通过 Python 编程可帮助用户完成烦琐的日常工作；
- 科学计算，Python 简洁的语法可以像使用计算器一样来完成科学计算；
- 快速原型，Python 省去了编译调试的过程，可以快速地实现系统原型；
- Web 编程，使用 Python 可以编写 CGI，而现在流行的 Web 框架也可以使用 Python 实现。

4. Python 是可扩展的

Python 提供了扩展接口，通过使用 C/C++可以为 Python 编写扩展。另外，Python 还可以嵌入到 C/C++编写的程序之中。在 C/C++编写的程序之中可以使用 Python 完成一些对于 C/C++实现起来较为复杂的任务。在某些情况下，Python 可以作为动态链接库的替代品在 C/C++中使用。

Python 可以很容易地被修改、调试，而不需要重新编译。使用 Python 也不必担心版本的问题。

5. Python 易学易用

Python 的语法十分简单，而且在 Python 中数据类型的概念十分模糊。在使用变量时无须事先声明变量的类型。

使用 Python 不必关心内存的使用和管理，Python 会自动地分配、回收内存。

Python 提供了功能强大的内置对象和方法。使用 Python 可以减少其他编程语言的复杂性。例如，在 C 语言中使用数十行代码实现的排序。而在 Python 中，可以通过列表的排序函数轻松完成。通过几天的学习，加上对 Python 模块函数的了解，便可以使用 Python 很快地编写出功能强大的脚本。

1.3 其他程序设计语言中的 Python

由于 Python 代码具有简明、方便和易读等特性，因此，大部分高级程序设计语言也提供了对 Python 的实现支持，如在 Java 平台中实现了 Jython、在.NET 平台中实现了 IronPython 等。

1. Jython

Jython，旧称 Jpython，是 Python 语言的 Java 实现。

Java 是一种面向对象的程序设计语言。Java 基本上是仿照 C++进行开发的，与 C++不同的是，Java 是完全面向对象的，在 Java 中，舍弃了 C++中容易导致问题的指针。另外，Java 提供了自动垃圾回收的功能，用于回收不再被使用的内存。Java 和 Python 一样，都是解释性语言，并且 Java 也是跨平台的，而且 Java 和 Python 都能将代码编译。但相对于 Python 而言，Java 则过于庞大和复杂。

Jython 是由 Jim Hugunin 使用 Java 对“原始” Python 的重写。由于 Jython 的出现，为了避免混淆，将通常所说的 Python 也称之为 CPython（因为其是由 C 编写的）。CPython 是相对于 Jython 而言的。Jython 的出现使得 Python 可以在 Java 环境中运行。Jython 和 CPython 的运行方式相同，在 Jython 下编写 Python 脚本和在 CPython 下编写 Python 脚本并没有太多的区别。

相对于 CPython 而言，Jython 要慢一些。这是因为 Java 是解释型的语言，Jython 本身首先要由 Java 解释器进行解释，而编写的 Python 脚本又需要由 Jython 解释器进行解释，这当然要比 CPython 慢一些。多数情况下，这并不是什么大问题。由于 Java 的流行，有大量的 Java 程序可以

使用。有了 Jython，就可以在 Python 中非常方便地使用 Java 丰富的链接库。除此之外，使用 Jython 还可以在 Java 中使用 Python 快速、简便地进行开发，充分使用 Python 的灵活和快速。

2. Python for .NET

.NET 是 Microsoft XML Web Services 平台，是微软所极力推崇的技术。使用 XML Web Services，不同的应用程序可以进行通信和数据共享。即使这些应用程序运行在不同的操作系统上，也不会影响它们之间的通信和数据共享。

随着.NET 的流行，出现了 Python for .NET 和 IronPython 这两种.NET 平台上的 Python。

Python for .NET 可以使用 Python 与.NET 进行交互，在.NET 中使用 Python 作为脚本语言，而且通过 Python for .NET，Python 也可以使用.NET 中的服务和组件。

3. IronPython

IronPython 也是 Python 编程语言在.NET 平台上的实现，由 Jim Hugunin（同时也是 Jython 创造者）所创造。IronPython 提供了和 Python 一样的交互式命令行。在交互式命令行下，可以使用 Python 访问所有的.NET 库。IronPython 除了对 Python 语言完全兼容外，还支持所有的.NET 类型库，并且继承了.NET Framework 的优点。通过 IronPython 可以使用 Python 来扩展.NET。

与 Python for .NET 相比，IronPython 的功能更加强大。IronPython 支持静态编译，通过静态编译，可以将 IronPython 程序编译成常规的.NET 的可执行文件。另外还可以将 IronPython 程序静态编译为.NET 动态链接库，所编译的动态链接库可以被 C#、VB.NET 等调用。

不管是 Python for .NET 还是 IronPython，都还不完善。但是，借助于开源的神奇力量，相信 Python 在.NET 上完善地使用只是时间上的问题。而且，微软的.NET 在开源社区也很受欢迎，甚至在 Linux 上也可以运行.NET 程序。

1.4 快速搭建 Python 开发环境

Python 可以运行在多种操作系统上，本书是以 Windows 为平台来介绍 Python。Python 官方网站提供了 Windows 下的安装程序，通过运行安装程序可以非常方便地安装 Python。除了安装 Python 以外，还应该选择用于编辑 Python 脚本的文本编辑器，提高编辑脚本的效率。因为在 Python 中，是使用缩进来表示语句块，因此所选择的文本编辑器最后具有自动缩进功能。

1.4.1 哪些系统中可使用 Python

本章前面已介绍过 Python 是跨平台的，可以运行在绝大多数的操作系统中。程序员平常用得比较多的操作系统主要包括以下几种：

- Windows
- Linux/UNIX
- Mac OS X
- OS/2

这些操作系统都可以使用 Python，因此，对于普通的用户而言，一般不需要担心自己所使用的系统不能运行 Python（绝大多数的流行操作系统都可以使用 Python）。本书以 Windows 7 操作

系统为平台，主要使用 Python 3.2 进行讲解（Python 现在最新版已是 3.4 了，但由于很多扩展库都还不支持最新版，因此，本书选用 Python 3.2 进行介绍）。

1.4.2 Python 的下载和安装

Python 的安装程序以及源代码都可以从其官方网站 http://www.python.org/获取。下面以 Windows 7、Python 3.2 为例，介绍在 Windows 下安装 Python 的过程，具体操作步骤如下。

step 1 从 Python 官方网站 http://www.python.org/下载 Python 在 Windows 下的安装程序 python-3.2.5.msi。

step 2 双击下载的安装程序进行安装，显示如图 1-2 所示界面。

step 3 如果系统中存在多个用户，而其他用户并不需要使用 Python，则可以选中【Install just for me】单选按钮，否则按照默认选项设置（所有用户都可使用 Python）。

step 4 单击【Next】按钮，显示如图 1-3 所示界面，设置安装路径。此处既可以按照默认安装路径，也可以根据需要选择 Python 的安装路径。



图 1-2 Python 安装程序

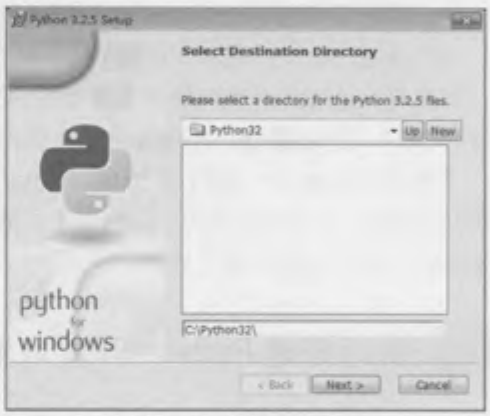


图 1-3 选择安装路径

step 5 单击【Next】按钮，进入选择安装模块界面，如图 1-4 所示。此处不需要修改，按照默认配置即可。

step 6 单击【Next】按钮，Python 安装程序将开始复制安装文件，如图 1-5 所示。



图 1-4 选择安装模块



图 1-5 复制安装文件

step 7 当 Python 安装程序复制完文件以后，将进入如图 1-6 所示界面。单击【Finish】按钮，完成 Python 安装。

step 8 安装完 Python 后，可以单击【开始】|【所有程序】|【Python 3.2】|【Python (command line)】命令，将弹出如图 1-7 所示的 Python 的交互式命令行界面。



单击【开始】|【运行】命令，在弹出的对话框中输入“python”命令，单击【运行】对话框的【确定】按钮，也可以打开 Python 的交互式命令行。



图 1-6 安装完成



图 1-7 Python 交互式命令行

如果在【运行】对话框中输入“python”命令没能打开 Python 的交互式命令行，则可能是由于 Windows 找不到 Python.exe 文件。这时，可以将 Python 的安装路径添加到 Windows 系统的“Path”系统变量中。具体操作步骤如下。

step 1 右击桌面上的【计算机】图标，在弹出的快捷菜单中选择【属性】命令，将弹出【系统】对话框，如图 1-8 所示。



图 1-8 【系统】对话框

step 2 在图 1-8 所示对话框左侧单击【高级系统设置】，将显示如图 1-9 所示的【系统属性】对话框，选择【高级】标签。

step 3 单击【环境变量】按钮，将弹出如图 1-10 所示的【环境变量】对话框。选中【用户变量】中的“Path”选项，单击【编辑】按钮，将弹出如图 1-11 所示的【编辑用户变量】对话框。在【变量值】文本框的末尾添加“;C:\Python32”，单击【确定】按钮。重启系统后即可。



图 1-9 【系统属性】对话框



图 1-10 【环境变量】对话框



图 1-11 【编辑用户变量】对话框

1.4.3 用 VS2008 编译 Python 源码

由于 Python 提供了源代码，因此程序员可以自己编译 Python。在 Windows 平台下可以使用 VC++6.0、Visual Studio 2013 等来编译 Python。对于 Python 3.2 的源代码，用 Visual Studio 2008（简称 VS 2008）进行编译最佳（并不是版本越高越好），下面介绍在 Windows 下编译 Python 的过程，具体步骤如下。

step 1 从 Python 官方网站下载 Python 的源代码压缩包 Python-3.2.5.tar.bz2。

step 2 将 Python 源代码解压至“D:\Python-3.2.5”，使用 Visual Studio 2008 打开其中的“PCbuild”目录下的“pcbuild.sln”文件，如图 1-12 所示是【解决方案资源管理器】中列出的 Python 各项目。

step 3 单击菜单【生成】|【批生成】命令，打开如图 1-13 所示的【批生成】对话框，将【平台】为“Win32”的项选中，单击右侧的【生成】复选框。

step 4 单击菜单【生成】|【生成解决方案】命令，VS2008 开始编译解决方案。

step 5 经过一段时间的编译、链接后，在 PCbuild 目录下将会生成相应的文件，其中 python.exe 是 Release 版的可执行文件，python_d.exe 是 Debug 版本的可执行文件。同时还会有 python32.dll 和 python32_d.dll 等文件，如图 1-14 所示。

step 6 双击 python.exe（或 python_d.exe）文件即可对 Python 交换环境，也可在命令行中执行 python.exe（或 python_d.exe）。

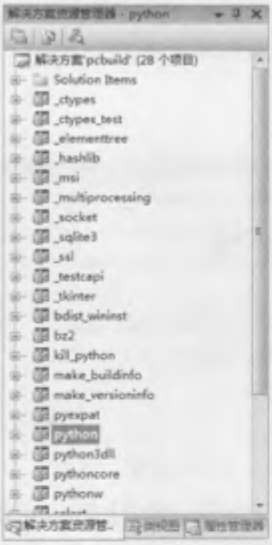


图 1-12 【解决方案资源管理器】对话框



图 1-13 【批生成】对话框

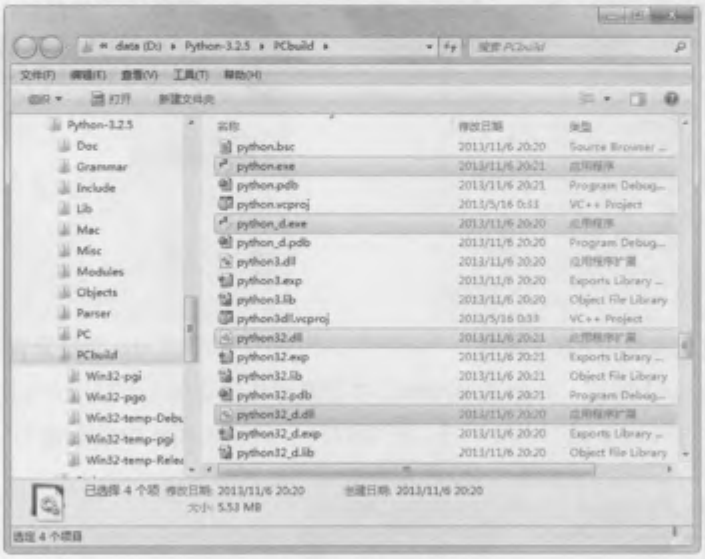


图 1-14 编译生成的文件

1.4.4 Python 开发工具：Vim

Vim（Vi IMproved）是由 Bram Moolenaar 编写的功能强大的文本编辑器。

Vi 是 UNIX 下经典的文本编辑器，也是每个 UNIX 系统标准配置的文本编辑器，Vim 是 Vi 的改进版本。Vi 只能在 UNIX 下使用，而 Vim 既可在 UNIX 下使用，也可在 Windows 下使用。Vim 和其他开源软件一样，具有非常好的可移植性。

Vim 提供强大的程序代码编辑功能，如自动缩进、代码折叠、语法高亮等。因此，可使用 Vim 作为 Python 的开发工具。

1. 安装 Vim

下面以 Windows 7 为例，介绍 Vim 的安装，具体步骤如下。

step 1 从 Vim 官方网站 http://www.vim.org/下载 Windows 版的安装程序 gvim74.exe。

step 2 双击安装程序，将显示如图 1-15 所示安装界面。

step 3 单击【是】按钮，进入【安装协议】界面，如图 1-16 所示。

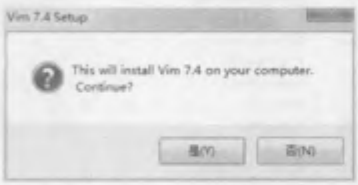


图 1-15 【Vim 安装】界面

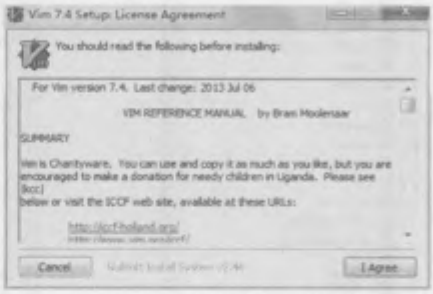


图 1-16 【安装协议】界面

step 4 单击【I Agree】按钮，进入【安装模块选择】界面，如图 1-17 所示。选中【Create .bat files for command line use】，可以在 Windows 的命令行下使用 Vim。

step 5 单击【Next】按钮，将显示如图 1-18 所示的界面，选择安装位置。

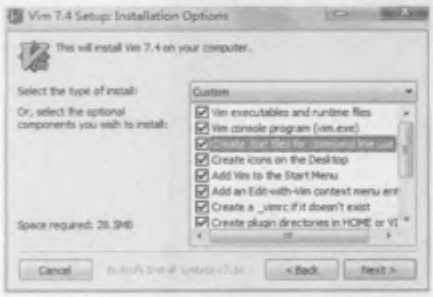


图 1-17 【安装模块选择】界面

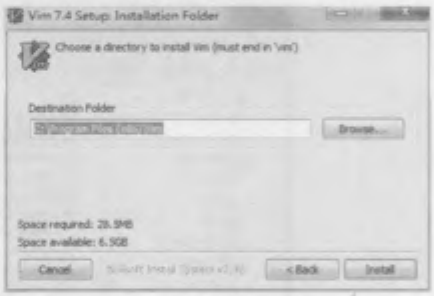


图 1-18 选择安装位置

step 6 单击【Install】按钮，开始安装 Vim。在安装过程中将弹出如图 1-19 所示的【命令行】窗口，这时可按回车键（也可不按，安装程序会自动关闭该命令行窗口）。当 Vim 安装程序复制完文件后将显示如图 1-20 所示界面。



图 1-19 【命令行】窗口

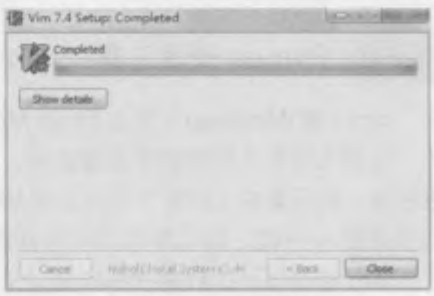


图 1-20 安装完成

step 7 单击【Close】按钮，将弹出如图 1-21 所示的【提示】对话框，提示是否阅读 README 文件，可以选择【否】按钮直接退出。

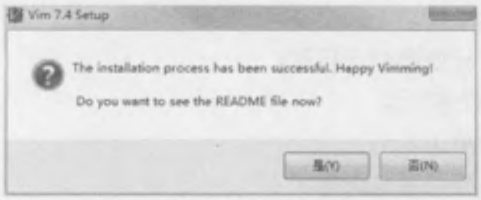


图 1-21 【提示】对话框

当完成 Vim 安装后，将在桌面创建 3 个快捷方式，其作用如下。

- gVim 7.4　　标准的 Vim。
- gVim Easy 7.4　　无模式的 Vim。
- gVim Read only 7.4　　制度模式的 Vim。

安装好 Vim 后，还需要打开 Vim 安装目录下的“vimrc_example.vim”文件，对 Vim 进行简单的设置。在“vimrc_example.vim”文件的最后添加配置命令“set nobackup”，可以不生成备份文件，添加“set nu”可以显示行号，添加“colo 配色方案名”可以修改默认配色方案。

Vim 编辑器界面如图 1-22 所示。



图 1-22 Vim 编辑器界面

2. Vim 命令简介

Vim 是有模式编辑器，它有两种模式：命令模式和编辑模式。在命令模式下输入的字符代表控制编辑器进行各种操作的命令，在编辑模式下则像使用其他文本编辑器（如 Windows 的记事本）一样，键盘输入的字符将出现在编辑窗口中。在编辑模式下按 Esc 键可以回到命令模式。

Vim 常用的命令有以下几个。

- ~　　转换大小写。
- A　　在行尾追加字符。
- a　　在光标之后追加字符。
- b　　向后移动一个单词，至单词首字母。

- B　向后移动一个单词，至单词首字母。
- cc　修改当前行。
- cw　修改单词。
- D　从光标处删除至行尾。
- dd　删除当前行。
- dw　删除单词。
- e　向前移动至单词尾。
- E　向前移动至单词尾。
- h　向左移动一个字符。
- i　在光标处插入字符。
- I　在行首非空白处插入字符。
- j　向下移动一行。
- J　合并两行。
- k　向上移动一行。
- l　向右移动一个字符。
- o　新建空白行。
- p　在光标前插入删除的字符。
- R　在光标处开始替换直到按 ESC 键。
- r　在光标处替换一个字符。
- s　删除光标处字符，并插入新字符。
- S　删除当前行，并插入新字符。
- u　撤销。
- V　进入块模式（在 Windows 下需要按 Ctrl+Q）
- v　进入行模式。
- w　向前移动一个单词，至词首。
- W　向前移动一个单词，至词首。
- x　删除光标处字符。
- yw　复制一个单词。
- yy　复制一行。

上述所列举的命令都是在命令模式下输入的命令字符。如果在这些命令前加上数字，则表示重复执行多少次命令。例如，在命令模式下输入“3dd”将删除从当前行开始的一共三行文本。

另外，在编辑模式下有几个对编程非常有用的命令，如 Ctrl+P（先按下 Ctrl 键再按下 P 键）命令或者 Ctrl+N 命令可以补全当前单词（前提是该单词已经在当前 Vim 编辑的文件中出现）。如图 1-23 所示，在第 23 行输入“button2 = Tk”，然后按 Ctrl+P，在 Vim 窗口下方可看到“关键字补全”的提示，同时，在“Tk”字符下方将出现一个列表，可从列表中选择一项进行补全。

如果所安装的 Vim 支持 Python，在编辑 Python 脚本时还可以通过先按 Ctrl+X，然后按 Ctrl+O 自动补全 Python 模块中的函数或者属性。如图 1-24 所示，在第 5 行导入 os 模块，在第 6 行中输入“os.pa”，然后先按 Ctrl+X，再按 Ctrl+O，将显示 os 模块中以“pa”开头的函数或属性。



图 1-23 自动补全当前单词



图 1-24 自动补全模块中的函数

在 Vim 7.4 中，如果不能使用 Ctrl+X、Ctrl+O 进行自补全提示，则可修改“C:\Program Files (x86)\Vim\vim74\ftplugin\”文件夹（若安装位置不同，这里的文件夹也不同）中的 python.vim 文件，在这个文件中找到以下内容：

```
setlocal omnifunc=pythoncomplete#Complete
```

修改为以下内容：

```
setlocal omnifunc=python3complete#Complete
```

如图 1-25 所示。

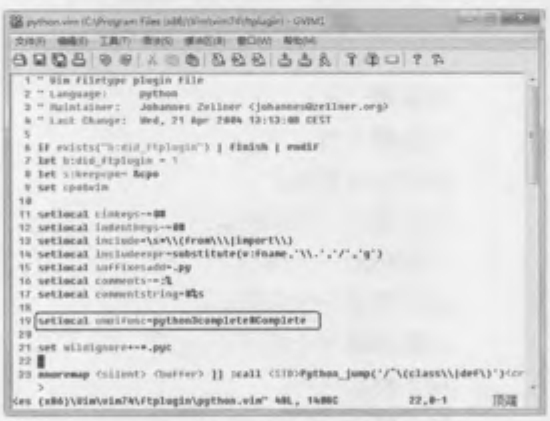


图 1-25 修改后的文件

Vim 还支持正则表达式，使用正则表达式可以完成非常复杂的查找替换工作。

1.4.5 Python 开发工具：Emacs

被讨论最多的文本编辑器是 Vim 和 Emacs，而且关于是选用 Vim 还是 Emacs 的争论从来没有停止过。

Emacs 被设计得“无所不能”，号称是世界上最强大的文本编辑器。与 Vim 不同，Emacs 不是有模式编辑器，使用 Emacs 就像使用 Windows 的记事本一样，但 Emacs 比 Windows 记事本的功能要强大得多。

可以从 http://ftp.gnu.org/gnu/emacs/windows/网站下载编译好的 Windows 安装程序，然后在

Windows 下进行安装。Emacs 安装程序是一个自解压的压缩文件，只需选择解压目录进行解压即可。解压完成后，运行 Emacs 所在目录下“bin”目录中的“runemacs.exe”文件，即可启动 Emacs 程序，如图 1-26 所示。



图 1-26　Emacs 文本编辑器

Emacs 中常用的命令如下。

- C-v　向后翻一页。
- M-v　向前翻一页。
- C-l　将当前行居中。
- C-f　向前移动一个字符。
- M-f　向前移动一个单词。
- C-b　向后移动一个字符。
- M-b　向后移动一个单词。
- C-n　向下移动一行。
- C-p　向上移动一行。
- C-a　移至当前行的第一个字符。
- M-a　移至当前所在句子的第一个字符。
- C-e　移至当前行的最后一个字符。
- C-p　移至当前所在句子的最后一个字符。
- M-<　移动到当前窗口的第一个字符。
- M->　移动到当前窗口的最后一个字符。
- C-x C-c　永久离开 Emacs。
- C-x C-f　读取文件到 Emacs。
- C-x r　以只读的方式打开一个文件。
- C-x C-q　清除一个窗口的只读属性。

- C-x C-s 保存文件到磁盘。
- C-x s 保存所有文件。
- C-x i 插入其他文件的内容到当前缓冲。
- C-x C-v 用将要读取的文件替换当前文件。
- C-x C-w 将当前缓冲写入指定的文件。
- C-s 向前查找。
- C-r 向后查找。
- C-M-s 规则表达式查找。
- C-M-r 反向规则表达式查找。
- M-p 选择前一个查找字符串。
- M-n 选择下一个查找字符串。
- C-d 向前删除字符。
- M-d 向前删除到字首。
- M-DEL 向后删除到字尾。
- M-0 C-k 向前删除到行首。
- C-k 向后删除到行尾。
- C-x DEL 向前删除到句首。
- M-k 向后删除到句尾。
- M-- C-M-k 向前删除到表达式首部。
- C-M-k 向后删除到表达式尾部。
- C-x r r 复制一个矩形到寄存器。
- C-x r k 删除矩形。
- C-x r y 插入刚刚删除的矩形。
- C-x r o 打开一个矩形，将文本移至右边。
- C-x r c 清空矩形。
- C-x r t 为矩形中的每一行加上一个字符串前缀。
- C-x r i r 从 r 缓冲区内插入一个矩形。
- C-x 1 删除所有其他窗口。
- C-x 2 上下分割当前窗口。
- C-x 3 左右分割当前窗口。
- C-x 0 删除当前窗口。
- C-M-v 滚动其他窗口。
- C-x o 切换光标到另一个窗口。
- C-x ^ 增加窗口高度。
- C-x { 减小窗口宽度。
- C-x } 增加窗口宽度。

需要说明的是，上面列出的命令中的“C”代表按下 Ctrl 键，“M”代表按下 Alt 键。例如，命令“C-v”是指按住 Ctrl 键不放，然后再按“v”键，而命令“C-x C-f ”则是指按住 Ctrl 键不放，

先按下“x”键，然后松开“x”键再按“f”键；而“M->”命令则指先按住 Alt 键不放，然后再按住 Shift 键不放，最后再按“.”，这是因为按住 Shift 键不放然后按“.”才是“>”。

1.4.6 Python 开发工具：PythonWin

前面所提到的 Vim 和 Emacs 两种文本编辑器各有特色，但上手较难。尤其是 Emacs 在操作的时候需要使用大量的快捷键，初学者很难记住。不过，一旦习惯，并且深入学习 Vim 和 Emacs 后，会发现它们确实非常强大。

在 Windows 下还有一个比较好用的 Python 脚本编辑器——PythonWin 所附带的编辑器。PythonWin 是 Python 在 Windows 下的扩展包，使用 PythonWin 可以让 Python 使用 Windows 系统的特性。

1. PythonWin 安装

在 Windows 下使用 Python 最好安装 PythonWin 扩展包。在 PythonWin 中提供了 Win32API 函数的封装，以及 MFC 类库的封装，通过 PythonWin 的相关模块可以在 Python 中直接调用 Windows 的 API 函数。PythonWin 的安装步骤如下。

step 1 从 PythonWin 官方网站 https://sourceforge.net/projects/pywin32/下载 PythonWin 的安装程序 pywin32-218.win32-py3.2.exe（注意文件名中的 py3.2 表示适用于 Python 3.2 版）。

step 2 双击安装程序后如图 1-27 所示。

step 3 单击【下一步】按钮，安装程序将自动搜索 Python 的安装路径，如图 1-28 所示。如果未能找到 Python 的安装路径，则需要检查 Python 的版本是否与 PythonWin 的版本相对应，或者重新安装 Python。



图 1-27 PythonWin 安装程序



图 1-28 自动搜索 Python 安装路径

step 4 单击【下一步】按钮，进入确认安装界面，如图 1-29 所示。

step 5 单击【下一步】按钮，PythonWin 的安装程序将开始复制装文件。当文件复制完成后，将出现如图 1-30 所示界面。单击【完成】按钮，即可完成 PythonWin 的安装。

2. PythonWin 编辑器简介

PythonWin 提供了一个简单易用的编辑器。在 PythonWin 中不仅可以交互式地运行 Python 命令，还可以编写 Python 脚本。单击【开始】|【所有程序】|【Python 3.2】|【PythonWin】命令，将打开 PythonWin 集成环境，如图 1-31 所示。PythonWin 将自动打开 Python 的交互式命令行窗口。单击【File】|【New】命令，可以新建 Python 脚本。



图 1-29 确认安装界面



图 1-30 安装完成界面

PythonWin 中还提供了自动补全功能，例如，当导入模块后，在模块名后输入“.”以后，PythonWin 将弹出一个列表窗口，使用方向键可以选择列表中的项目，按下 Tab 键可以补全，如图 1-32 所示。

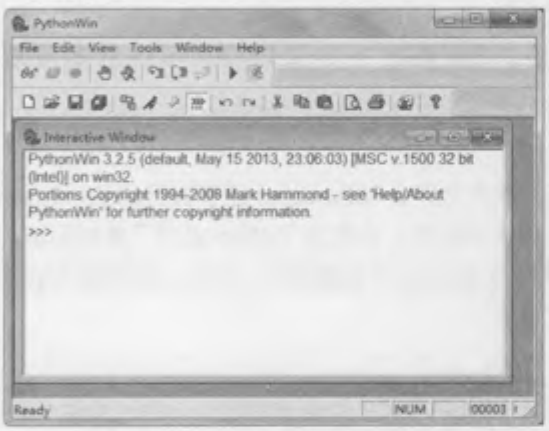


图 1-31 PythonWin 集成环境

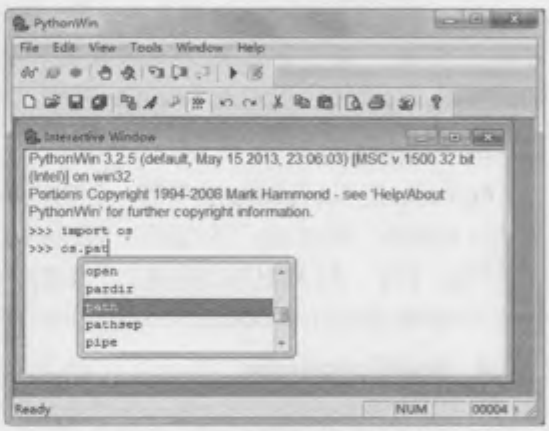


图 1-32 模块函数自动补全

如果所输入的变量已经输入过，则可以按“Alt+/”键来自动补全。另外，在 Python 的交互式命令行下，可以按 Ctrl 键加向上的方向键（或者向下的方向键）不重复输入前面所输入的命令。

1.4.7 其他的 Python 开发工具

除了上述的 Vim、Emacs 和 PythonWin 外，开发 Python 时还有很多的编辑器或者集成开发环境可供选择。另外，Python 自身还附带了一个图形化的交互式环境 IDLE，使用 IDLE 也可以编辑 Python 脚本。

IDLE 使用 Tkinter 创建 GUI 界面。安装好 Python 以后，单击【开始】|【所有程序】|【Python 3.2】|【IDLE（Python GUI）】命令，可以打开 IDLE，如图 1-33 所示。

下面简单介绍一些可以作为 Python 集成开发环境的软件。

1. BlackAdder

BlackAdder 可以运行在 Windows 和 Linux 系统中。BlackAdder 为用户提供了个人版和专业版，分别面向个人用户和商业用户。



图 1-33 IDLE 界面

2. Wing IDE

Wing IDE 是使用 Python 编写的，可以运行在 Windows 和 Linux 系统中。Wing IDE 提供一个源码分析器、浏览器以及文本编辑器和调试器。WingIDE 为共享软件，可以从其网站 http://www.archaeopteryx.com/wingide 下载试用版。

3. Komodo

Komodo 是 ActiveState 提供的一个 Python IDE，除此之外，ActiveState 还有自己的 Windows 版本的 Python。Komodo 可以运行在 Windows 和 Linux 系统中，不仅为 Python 提供了集成环境，还为 Perl、PHP、Tcl 和 HTML 等提供了集成开发环境。Komodo 为共享软件，可以从其网站 http://www.activestate.com/products/komodo_ide/下载试用版。

4. Boa Constructor

Boa 是使用 Python 和 wxPython 编写的跨平台的 Python IDE，它提供可视化的编程和操作框架，能方便地进行程序的设计。Boa 提供了对象浏览器，并提供各种资源的视图，在 Boa 中还包含一个 HTML 文档生成器、调试器和完整的帮助系统。Boa 还提供对 zope 的支持。Boa 是免费的，可以从其官方网站 http://boa-constructor.sourceforge.net/下载，在安装之前要先安装合适版本的 Python 和 wxPython。

5. PyDev

PyDev 是 Eclipse 中的 Python 开发插件。Eclipse 是著名的跨平台的自由集成开发环境，主要用来进行 Java 语言开发。Eclipse 的本身只是一个框架平台，但是众多插件的支持使得 Eclipse 拥有其他功能相对固定的 IDE 软件很难具有的灵活性。许多软件开发商都以 Eclipse 为框架开发自己的 IDE。

可以从 PyDev 的官方网站 http://pydev.org/下载 PyDev，然后在 Eclipse 中安装 PyDev 插件，即可使用 Eclipse 开发 Python。

6. Eric3

Eric3 是一个功能强大的 Python IDE，它基于 QScintilla 编辑器组件，使用 Python 和 PyQt 编写。Eric3 中集成项目管理工具，可以生成类 UML 的图表，还包含一个功能强大的 Python 调试器。Eric3 可以从其官方网站 http://eric-ide.python-projects.org 下载。

7. DrPython

DrPython 是一个跨平台的高可配置的程序开发环境，使用 Python 编写。DrPython 基于 wxPython 和 Scintilla 库。DrPython 可以从其官方网站 http://drpython.sourceforge.net/下载。

8. SciTE

SciTE 是一个基于 SCIntilla 和 GTK+开发的，可以运行在 Windows 和 Linux 系统中。SciTE 支持语法高亮、代码折叠等，并可以将代码导出为 HTML、RTF 和 PDF。SciTE 可以从其官方网站 http://scintilla.sourceforge.net/下载。

1.5 第一个 Python 程序

在 Windows 下有多种方式可以运行 Python 脚本，也可以直接在 Python 的交互式命令行下一句一句地编写运行 Python 脚本。当代码很多的时候，则应该在文本编辑器中将代码编辑好，然后再运行。

1.5.1 从“Hello，Python!”开始

如果读者学习过 C 语言，应该使用 C 语言编写过第一个“Hello，World”程序。使用 Python 编写这样的一个程序仅需要一行代码，如下所示。

```
Print('Hello,Python!')
```

打开文本编辑器（或本章前面介绍的任意一款开发工具软件），输入上述代码（如图 1-34 所示是在 Vim 中输入以上代码的界面），然后将其保存为“HelloPython.py”。



图 1-34 在 Vim 中编写第 1 个 Python 程序

对于 Python 程序文件，在 Windows 下可以直接通过双击来运行。双击上面编写的程序文件“HelloPython.py”，可以看到一个命令行窗口出现，然后又关闭，由于很快，可能看不到输出内容。为了能看到脚本的输出内容，可以按以下步骤进行操作。

step 1 单击【开始】|【运行】命令，在【打开】文本框中输入命令“cmd”，如图 1-35 所示。

step 2 单击【确定】按钮，打开 Windows 的命令行窗口，如图 1-36 所示。

step 3 进入“HelloPython.py”所在的目录，如“E:\Python”，在命令行提示符下输入“HelloPython.py”或者“python HelloPython.py”，然后按回车键即可运行“HelloPython.py”脚本，如图 1-36 所示。



图 1-35 “运行”对话框



图 1-36 运行“HelloPython.py”脚本

1.5.2 Python 的交互解释器

Python 还提供了交互式命令行，可以一边输入脚本，一边运行脚本。通过单击【开始】|【所有程序】|【Python 3.2】|【Python（command line）】命令，可以打开 Python 的交互式命令行，在命令行中输入“print('Hello,Python!')”，然后按【Enter】键，将得到如图 1-37 所示的结果。



图 1-37 Python 交互式命令行

如果安装了 PythonWin，则可以使用 PythonWin 中的 Python 交互式命令行。在 PythonWin 中可以方便地粘贴、复制代码，还可使用 PythonWin 提供的自动补全功能，方便程序输入。单击【开始】|【所有程序】|【Python 3.2】|【PythonWin】命令，打开 PythonWin。单击【File】|【Open】命令，打开“HelloPython.py”脚本，如图 1-38 所示。



图 1-38 在 PythonWin 中打开“HelloPython.py”脚本

单击工具栏中的【Run】图标，弹出如图 1-39 所示的对话框，单击【OK】按钮，即可运行“HelloPython.py”脚本，

此时 PythonWin 的交互式窗口输出了文字“Hello, Python!”，如图 1-40 所示。

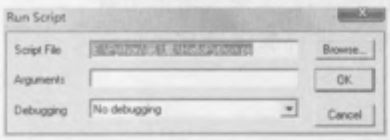


图 1-39 运行“HelloPython.py”脚本

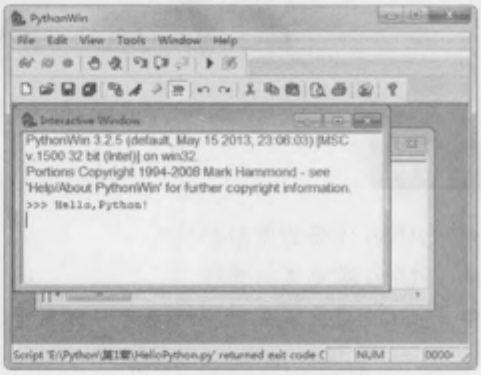


图 1-40 输出脚本

1.6 本章小结

本章首先简要介绍了 Python 及 Python 的优点，以及其他程序设计语言中集成的 Python。接着详细介绍了 Python 的下载和安装，如果下载的是源码，还可用本章介绍的方法通过 VS2008 进行编译。然后介绍了编写 Python 的常用开发工具的使用（包括 Vim、Emacs 和 PythonWin），在 Windows 操作系统中使用 Python，可能更多情况下是使用 PythonWin。最后还引导读者编写第一个 Python 程序。

本章主要是搭建 Python 环境，为下一章学习 Python 基本语法做好准备。

◆ ◆ ◆

第 2 章 Python 起步必备

本章包括

- Python 代码的组织形式
- Python 对中文的支持
- Python 对大整数的支持
- Python 的基本输入输出函数
- Python 进行算术运算

Python 的语法非常简单，容易学习和掌握。在详细介绍语法之前，本章先对 Python 做整体地介绍。主要介绍脚本的结构、中文的使用等一些容易导致脚本出错，但对于初学者而言却又不容易解决的问题，以免初学者在使用 Python 时被一些简单的问题所困扰。

2.1 Python 代码的组织形式

学习任何一种程序设计语言，首先都应了解代码的组织形式、注释方式等基础内容。本节就先介绍 Python 代码的组织形式和注释方式。

2.1.1 用缩进来分层

Python 脚本的结构非常清晰，在 Python 中，不像 C 语言那样使用花括号来表示语句块，而是使用代码缩进来表示分层。

在 Python 中，代码缩进一般用在函数定义、类的定义以及一些控制语句中。一般来说，行尾的 ":" 表示代码缩进的开始。如下所示的代码为在判断语句中使用缩进。

```
if a > b:
   print(a)                # 代码缩进，如果 a>b，则执行 print a
else:
   print(b)                # 代码缩进，如果 a<=b，则执行 print b
```

下面是一个比较复杂的代码缩进的例子。

```
if a > b:
   if a == 1:              # 代码缩进
      print(a)             # 代码缩进，即缩进嵌套
   else:
      if a == 0:
         print(a)
      else:
         pass
elif a == b:
   print(a,b)
else:
   print(b)
```

需要注意的是，处于同一级的代码缩进，其缩进量要保持一致，如下所示，代码缩进量不一致，将导致错误。

```
if a > b:
   if a == 1:
      print(a)
   else:
      if a == 0:
         print(a)
      else:
         pass
elif a == b:
  print a                    # 此处代码和下一句的代码缩进不一致，脚本运行错误
   print(b)
else:
   print(b)
```

很多人习惯使用 Tab 键进行缩进，笔者不建议采用这种方式。最好通过空格的形式缩排代码，每一层向右缩进 4 个空格。

2.1.2 两种代码注释的方式

注释是程序中必不可少的部分，可方便程序员之间进行交流。

在 Python 中，注释语句以字符“#”开始，位于“#”之后的语句不被执行。字符“#”仅注释其所在的行。

在 Python 中，如果进行大段的注释，可以使用三个单引号“'''”或者三个双引号“"""”将注释内容包围。如下所示为两种不同方式的注释的示例。

```
'''
三个单引号包围的注释
该段代码判断 a，b 值的大小
并根据不同的情况输出
如果 a 大于 b 则输出 a
如果 a 小于等于 b 则输出 b
'''
if a > b:                  # 判断 a 和 b 的大小
   print(a)                # 输出 a

else:
   print(b)                # 输出 b
"""
三个双引号包围的注释
代码判断结束
Print(a)
上边的语句不会被执行
"""
```

2.1.3 Python 语句的断行

在 Python 中，一般来说一条语句占用一行，在每条语句的结尾处不需要使用 C 语言中的“;”

来作为结束标志。当然，在 Python 中也可以使用 ";" 将两条语句写在一行。

另外，如果缩进语句块中只有一条语句，也可以直接将这条语句写在 ":" 之后，如下所示。

```
if a > b:print(a)                    # 缩进语句写在冒号之后
else:print(b)
print(a);print(b)                    # 使用分号将两条语句写在同一行
```

在 Python 中，单引号和双引号没有区别，都可以用来包围字符串。另外，单引号中的字符串中可以包含双引号，双引号中的字符串也可以包含单引号，而不需要使用转义字符，代码如下所示。

```
a = "What's your name"
b = 'I say:"What is your name?"'
```

另外三个单引号或者三个双引号所包围的字符串（可以为多行）不仅可以作为注释，还可以作为格式化的字符。当使用 Python 中的 "print" 函数输出这些字符串时，其格式将保持不变，代码如下所示。

```
a = '''
这是格式化的字符
    此处的缩进将输出
在这也可以使用'
或者"
不影响
"""
当然还有三个双引号
"""
'''
b = """
这是三个双引号包围的
格式化    字符
'''
"""
```

在 Python 中，如果语句较长，需要分成几行书写，则可以使用 "\" 来进行续行，也可用一对圆括号来将一条语句写成几行，代码如下所示。

```
# 使用 "\" 续行
# 需要注意的是 "\" 之后不能有任何字符
# 不能在 "\" 之后使用 "#" 注释
c = a * 2\
    + b\
    - b\
    *3
# 使用圆括号包围分成多行的语句
# 在语句中可以使用 "#" 注释
c = ( a *
    b - 1
    + 3
    /
    2)
```

需要说明的是，在 Python 脚本中所有语句中的标点符号都是英文标点符号。在编写 Python 脚本时，最好将输入法切换到英文，避免输入中文标点符号导致脚本运行错误。不过，在字符串和注释中可以使用中文标点。

2.2 Python 的基本输入输出函数

为了与用户进行交互，所有的程序设计语言都提供了输入输出功能，Python 当然也不例外。通过输入函数，Python 程序在执行过程中可接收用户输入的数据；通过输出函数，Python 可以将程序处理结果显示出来，让用户观察到。

2.2.1 接收输入的 input 函数

Python 中的基本输入函数是 input（在 Python 3 中已不再使用以前的“raw_input”语句了）。input 函数将用户输入的内容作为字符串形式返回，如果想要获取数字，可以使用“int”函数将字符串转为数字。如下所示的代码在 Python 的交互式命令行中运行。

```
>>> input('Input your name:')              # 使用 raw_input 提示输入
Input your name:bluebanboom                # bluebanboom 为用户输入
'bluebanboom'
>>> name = input('Input your name:')       # 将用户输入赋值给 name
Input your name:bluebanboom
>>> print(name)                            # 输出 name
bluebanboom
>>> year = input('The year:')              # 获取输入
The year:2013
>>> print(year)
2013
>>> year + 1                               # year 加 1，这里导致出错，因为 year 为字符串型
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> int(year) + 1                          # 使用 int 函数将 year 转换成整型
2014
```

需要说明的是，位于“>>>”命令提示符之后的内容为用户输入的语句。如果语句前没有“>>>”命令提示符，则表示该语句为 Python 的输出。但是由于使用了“input”函数，因此在“input”函数的提示之后需要用户输入。如果在 PythonWin 的交互式命令行中使用了“input”函数，则将弹出如图 2-1 所示对话框。在文本框中输入内容后，单击【OK】按钮即可。

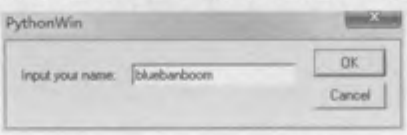


图 2-1 raw_input 对话框

在 Python 中，除了 int 函数外，还有以下用于类型相互转换的函数。

- float　　将字符串或者整数转换为浮点数。

- str　　将数字转换为字符串。
- chr　　将 ASCII 值转换为 ASCII 字符。
- hex　　将整数转换为十六进制的字符串。
- long　　将字符串转换为长整型。
- oct　　将整数转化为八进制的字符串。
- ord　　将 ASCII 字符转化为 ASCII 值。

2.2.2 输出内容的 print 函数

Python 中的基本输出函数是“print”（在 Python 3 之前版本称为 print 语句），在第 1 章的“HelloPython.py”脚本中已经使用了“print”函数。使用“print”函数可以输出 Python 中的所有数据类型的值，而不需要事先指定要输出的数据类型。如果自定义了某一新的类型（或者类），则可以通过重载“__repr__”，让“print”函数支持对该自定义类型的输出。如下所示的代码可在 Python 的交互式命令行中运行。

```
>>> a = 0
>>> print(a)                          # 输出整型
0                                     # 输出内容
>>> b = 1
>>> print(a+b)                        # 输出表达式的值
1
>>> print(b)                          # 输出 b 的值
1
>>> s = 'Hello'                       # 定义字符串
>>> print(s)                          # 输出字符串
Hello
>>> l = [1, 2, 3]                     # 定义列表
>>> print(l)                          # 输出列表
[1, 2, 3]
>>> t = ('a', 'b', 'c')               # 定义元组
>>> print(t)                          # 输出元组
('a', 'b', 'c')
>>> print(l,t)                        # 同时输出列表和元组
[1, 2, 3] ('a', 'b', 'c')
>>> print(l, '\n', t)                 # 使用换行符
[1, 2, 3]
('a', 'b', 'c')
>>> for i in t:                       # 循环输出
...     print(i)
...                                   # 在这一行的缩进处按一下回车键，表示缩进结束
a
b
c
```

上述代码中使用了缩进，在交互式命令状态下，用“...”表示缩进开始。如果使用的是 Python 自带的交互式命令行，则需要注意进行缩进；而如果是在 PythonWin 的交互式命令行中，则不需要，因为 PythonWin 提供了自动缩进。如果缩进的语句已经结束，则只需按一下回车键即可表示缩进结束。

2.3 Python 对中文的支持

在 Python 2.x 时，需要经过相关的设置才能支持中文。然而，从 Python 3 开始，对中文的支持就很全面了。本节首先介绍在 Python 3 之前的版本应该如何设置，以使其支持中文，然后再介绍 Python 3 对中文的全面支持。

2.3.1 Python 3 之前版本如何使用中文

在 Python 2.x 中可以使用中文，但需要对中文进行处理。在 Python 2.x 中，显示中文主要是字符编码的问题，如果处理不好将导致乱码。

在计算机中，字符是以数字来表示的。字符是通过字符编码将其转化为数字，以让计算机能够对其识别。最早的时候，计算机仅支持英文字符，也就是 ASCII 字符。ASCII 字符包含大小写字母，以及一些标点和其他的字符，使用一个字节来表示。但对于中文字符来说，使用一个字节显然不能满足需求。为了能够在计算机中表示所有的中文字符，中文编码采用两个字节表示。如果中文编码和 ASCII 混在一起，就会导致解码错误，从而产生乱码。采用两个字节的中文编码标准有 GB2312、GBK、BIG5 等。

为了将各种不同的语言都包含在统一的字符集中，满足国际间的信息交流，国际上制订了 UNICODE 字符集。UNICODE 字符集包含世界上所有语言字符，这些字符具有唯一的编码。通过使用 UNICODE 字符集可以满足跨语言的文字处理，有效地避免乱码的产生。

在 Python 中，可以在各种编码间相互转换。如果在交互式命令中使用中文，即便不做处理，一般也不会出现乱码。如果在 ".py" 文件中使用了中文，则需要在文件的第一行使用如下语句指定字符编码集。

```
# -*- coding:UTF-8 -*-
```

其中，UTF-8 表示使用 UTF-8 编码，也就是 UNICODE 字符集。使用上述语句，仅指明脚本中包含非 ASCII 字符，而并未将字符编码转换为 UTF-8 编码。如果要将字符编码改为 UTF-8，则需要在保存的时候选择保存成 UTF-8 的格式。在记事本中可以通过选择【文件】|【另存为】命令，在弹出的 "另存为" 对话框中的【编码】列表框中选择 "UTF-8"，如图 2-2 所示。



图 2-2 选择文件编码

在 Vim 中可以使用如下命令设置文件的编码。

```
:set fileencoding=UTF-8
```

需要注意的是，在有些情况下 Vim 选择 UTF-8 编码后容易导致乱码，这时，可以通过 Windows 的记事本将文件重新保存为 UTF-8 格式。

除了使用 UTF-8 编码以外，还可以使用 CP936、GB2312、ISO-8859-1 等来指定字符编码，从而在 Python 脚本中使用中文。需要注意的是，如果在命令行界面中输出中文字符，则需要设定编码为 CP936。例如，下面所示的脚本，将字符编码设定为 UTF-8，并保存成 UTF-8 编码格式，运行脚本后输出如图 2-3 所示。

```
# -*- coding:utf-8 -*-
# file: Chinese.py
#
Chinese = '''
在 Python 中使用中文
需要注意字符编码的问题
可以使用的字符编码有如下几种：
UTF-8、CP936、GB2312、ISO-8859-1。
'''
print Chinese
```

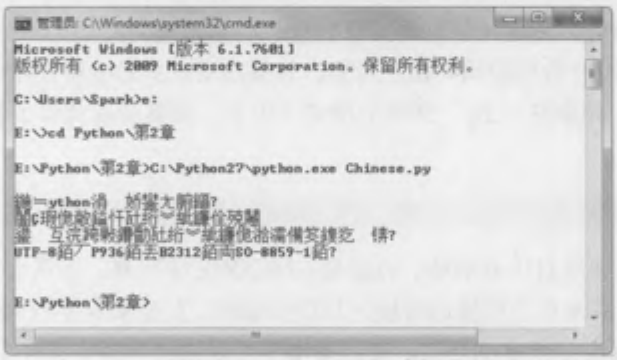


图 2-3　命令行乱码

这是因为 Windows 的命令行中采用的是 CP936 编码，而在上述脚本中采用的是 UTF-8 编码，因此导致乱码。解决的方法是，在脚本中使用 decode 和 encode 函数对字符重新解码编码，或者，不将其保存成 UTF-8 格式。如果仍想采用 UTF-8 编码格式，则可以将脚本修改成如下所示。

```
# -*- coding:utf-8 -*-
# file: Chinese.py
#
Chinese = '''
在 Python 中使用中文
需要注意字符编码的问题
可以使用的字符编码有如下几种：
UTF-8、CP936、GB2312、ISO-8859-1。
'''
print Chinese.decode('utf-8').encode('cp936')
```

注意，以上代码都是在 Python 2.x 中执行的，如果是在 Python 3 中执行，则 print 语句需要改为函数形式。

2.3.2 更全面的中文支持

随着 Python 3 的推出，Python 对中文的支持也变得非常全面。在 Python 3 中，源文件默认是使用 UTF-8 编码，这样一来，不但可方便地在源代码的字符串中使用中文，而且变量名也可以使用中文命名。例如，在 Python 交互命令状态下可执行以下代码，将变量名设置为“中国”。

```
>>> 中国='China'
>>> print(中国)
China
```

在 Python 3 中执行 2.3.1 节中的代码，则不需要来回的编码和解码，直接使用 print 函数即可输出变量 Chinese 中的内容。因此，可将 2.3.1 节中的代码修改为以下形式（保存为 Chinese3.py）：

```
# file: Chinese3.py
#
Chinese = '''
在 Python 中使用中文
需要注意字符编码的问题
可以使用的字符编码有如下几种：
UTF-8、CP936、GB2312、ISO-8859-1。
'''
print(Chinese)
```

在 Windows 命令窗口中执行以上代码，结果如图 2-4 所示。



图 2-4 Python3 对中文的支持

另外，在 Python 3 中，字符串对象没有 decode 和 encode 方法。

2.4 简单实用的 Python 计算器

在 Python 交互命令环境下，我们还可以将其作为一个计算器来使用，直接输入需要计算的表达式，Python 即可快速计算出结果。并且，由于 Python 提供了功能丰富的数学运算函数，因此，

可进行各种数学运算。另外，Python 还直接支持大整数的计算。下面就来演示 Python 的这些计算功能。

2.4.1 直接进行算术运算

由于 Python 具有交互式的命令行，因此在交互式命令行下，可以使用 Python 完成基本的数学运算，只需要在命令行状态下输入算式，Python 即可输出计算结果。以下代码在 Python 交互式命令行中运行。

```
>>> 3*5/2                    # 结果为整数
7
>>> 3.0*5.0/2                # 结果为小数
7.5
>>> 3.0+5.0/2
5.5
>>> (3.0+5.0)/2              # 使用括号改变运算符优先级
4.0
>>> 2**3                     # 求 2 的 3 次方
8
>>> 2**8                     # 求 2 的 8 次方
256
```

2.4.2 math 模块提供丰富的数学函数

Python 中提供了很多内置模块（有关模块的概念在第 4 章中介绍），其中名为 math 的模块提供了丰富的数学函数，在提交给 Python 运算的算式中可包含这些数学函数。其中主要的函数如下。

- sin(x)　　求 *x* 的正弦。
- cos(x)　　求 *x* 的余弦。
- asin(x)　　求 *x* 的反正弦。
- acos(x)　　求 *x* 的反余弦。
- tan(x)　　求 *x* 的正切。
- atan(x)　　求 *x* 的余切、反正切。
- hypot(x, y)　　求直角三角形的斜边长度。
- fmod(x,y)　　求 *x*/*y* 的余数。
- ceil(x)　　取不小于 *x* 的最小整数。
- floor(x)　　取不大于 *x* 的最大整数。
- fabs(x)　　求绝对值。
- exp(x)　　求 *e* 的 *x* 次幂。
- pow(x,y)　　求 *x* 的 *y* 次幂。
- log10(x)　　求 *x* 的 10 底对数。
- sqrt(x)　　求 *x* 的平方根。
- pi　　π 的值。

下面所示的代码使用了 Python 中的 math 模块进行数学运算（在使用这些数学函数之前，应使用 import math 将模块导入）。

```
>>> import math
>>> math.cos(0.5)
0.87758256189037276
>>> math.sin(math.pi)
1.2246063538223773e-016
>>> math.sin(60)
-0.30481062110221668
>>> math.tan(1)
1.5574077246549023
>>> math.sqrt(9)
3.0
>>> math.log10(2)
0.3010299956639812
>>> math.log10(100)
2.0
>>> math.asin(0.5)
0.52359877559829893
>>> math.pow(2,8)
256.0
```

另外，在 Vim 中也可以使用 Python 进行计算。例如，输入以下命令可以在 Vim 的命令缓冲区中显示求 3 的 5 次方。

```
:py print 3**5
```

在 Vim 中，可以通过以下命令导入 math 模块，并使用其中的函数进行计算。

```
:py import math
:py print(math.sin(1))
```

2.4.3 Python 对大整数的支持

在大多数程序设计语言中，保存整数的变量都有一个值的区域，当超过该区域后，就无法保存更大的数。例如，在 C 语言中，int 类型的变量使用 4 字节保存值，保存数据的范围为 -2 147 483 648 ~ 2 147 483 647。如果计算结果为更大的值，则无法保存，即使使用 long 类型，仍然有一个范围限制。这样，就无法进行如阶乘、大的幂运算等，要保存这些结果值很大的数据，就需要另外编写大整数处理程序。

幸运的是，在 Python 中，直接提供了对大整数的支持。我们可以直接调用。例如，在交互命令状态下执行下面的幂运算，可得到一个很大的整数。

```
>>> 99**99
369729637649726772657187905628805440595668764281741102430259972423552570455277
523421410650010128232727940978889548326540119429996769494359451621570193644014
418071060667659301384999779999159200499899
```

在上面的运算中，两个星号表示进行幂运算。从上面的结果可看出，计算的结果有 198 位，

远远超过了普通整型变量的表示范围，但是，Python 处理起来没有压力，也很简单。

2.5 本章小结

本章主要介绍了 Python 脚本的结构和基本语法，首先介绍了 Python 代码的组织形式，基本输入输出函数的使用。接着讲解了 Python 对中文的支持，这在 Python 3 中已得到很好的解决，不需要进行过多的设置；但是，对 Python 2.x 来说，还需按本章介绍的方法进行设置。最后，本章介绍了 Python 算术运算的能力，特别是对大整数的支持，使用户可以很方便地进行任意大小数据的运算。

在下一章中，将进一步学习 Python 的各种数据类型和流程控制语句。

◆ ◆ ◆

第 3 章　Python 数据类型与基本语句

本章包括

- Python 数据类型：数字
- Python 数据类型：列表和元组
- Python 数据类型：文件
- Python 数据类型：字符串
- Python 数据类型：字典
- Python 的流程控制语句

在使用 Python 进行程序设计之前，应了解 Python 提供了哪些数据类型，有哪些结构控制语句。对于一般的 Python 脚本来说，使用 Python 内置的数据类型就可以完成绝大多数工作，基本不必考虑自己重新定义数据类型或是数据结构。而在程序结构控制方面，Python 提供了三种基本的语句来控制分支和循环，可以很好地完成程序流程控制。

3.1　Python 数据类型：数字

数据类型是程序的基础，程序设计的本质就是对数据进行处理。由于 Python 有设计良好的数据类型，以及丰富的内置函数，因此使得 Python 脚本对数据的处理变得十分简单。大多数情况下，程序员只需使用内置数据类型就足够了。

数字是程序需要处理的最基本的数据类型，任何编程语言都提供了对数字类型的支持。正如第 2 章所介绍的，在 Python 中可以使用任意大的数，而不用担心溢出。

3.1.1　整型和浮点型

在 Python 2.x 中，数字类型共有 4 种，分别是整数（int）、长整数（long）、浮点数（float）和复数（complex），如表 3-1 所示。

表 3-1　Python 2.x 中的数字类型

类型	描述
整数（int）	一般意义上的数，包含八进制（以数字 0 开头）及十六进制（以 0x 开头），如 2007、-2007、07（八进制）和 0xAB（十六进制）等
长整数（long）	无限大小的数，在其结尾添加小写字母 l 或者大写字母 L，如 2007000000000000000L
浮点数（float）	小数，或者用 E 或 e 表示的幂，如 2.7、1234e+10、1.5E-10
复数（complex）	复数的虚部以字母 j 或 J 结尾，如 1+2j、2.2+2.0J

在 Python 3 中，已经没有 long 这种数字类型，整数就只有 int 这一种类型。不过，这时的 int 类型的作用与 Python 2.x 中的 long 相同，也就是说，在 Python 3 中，int 类型可保存任意大小的整数。

作为动态类型的语言，在 Python 中使用数字无须事先声明其类型，下面是一些演示代码。

```
>>> a = 1                          # 将 a 赋值为 1，整数
>>> b = 12.5                       # 将 b 赋值为 2，浮点数
>>> a + b                          # 计算 a + b
13.5
>>> c = 20070000000000000  # 长整数(在 Python 2.x 中，需要在长整数后面加上大写字母 L 或小
                           # 写字母 l)
>>> c
20070000000000000
>>> d = 200700000000000000 # 长整数
>>> d
200700000000000000
>>> d - c                          # 计算 d - c
180630000000000000
>>> d + b
2.007e+017                         # 浮点数
>>> 2.30 - 1.30
0.9999999999999998                 # 结果并不为 1.00，由浮点数的精度所导致
>>> 2.3 - 1
1.2999999999999998                 # 同样，这里的整型数字 1 被转换成浮点数进行运算
>>> 0o7 + 0o5                      # 八进制(第 1 个字符为数字 0，第 2 个字符为字母 o。
                                   # 在 Python 2.x 中，八进制数直接在数字前加 0 即可，不用加字母 o)
12                                 # 输出为十进制
>>> 0x7 + 0xa                      # 十六进制
17                                 # 输出为十进制
>>> print('%o' % ( 0o7 + 0o5)) # 输出八进制（单引号中的是字母 o，不是数字 0）
14
>>> print('%x' % ( 0x7 + 0x5)) # 输出十六进制
c
>>> m = 9 + 3j                     # 复数形式
>>> n = 15 - 2j
>>> m + n                          # 复数运算
(24+1j)
```

3.1.2 运算符

与其他高级程序设计语言一样，Python 也提供了丰富的运算符，以方便对数据进行运算。在 Python 中，除了基本的算术运算符，如加、减、乘、除、取余等运算符外，还有逻辑运算符，如位移动、位逻辑运算等，如表 3-2 所示。

表 3-2 Python 运算符

运算符	描述
**	乘方运算符
*	乘法运算符
/	除法运算符
//	整除运算符
%	取余运算符
+	加法运算符
-	减法运算符

续表

运算符	描述
\|	位或
^	位异或
&	位与
<<	左移运算
>>	右移运算

在复杂的表达式中往往使用多个运算符，表达式的计算顺序由运算符的优先级确定。在表达式中，先进行运算符优先级高的计算，对于同级运算符从左至右依次计算。

表 3-2 所示的运算符越往下优先级越低，乘方运算符的优先级最高，乘法运算符、除法运算符、取余运算符的优先级相同，其优先级较乘方运算符次之；加法运算符和减法运算符属同级运算符，它们的优先级较上述运算符次之；剩下的逻辑运算符属于同级运算符，它们的优先级最低。如果要使优先级低的运算符具有高优先级，则可以使用括号将表达式括起来。

下面代码演示了运算符的使用。

```
>>> 2 ** 5                  # 乘方运算，求解 2 的 5 次方
32
>>> 2 ** 0                  # 求解 2 的 0 次方
1
>>> 3 * 2                   # 乘法运算
6
>>> 4 / 2                   # 除法运算(在 Python 2.x 中，结果将为整数 2)
2.0
>>> 7 / 2
3.5                         # 在 Python 2.x 中，结果将被取整数，得到 3
>>> 7 // 2                  # 整除运算，结果将被取整数，得到 3
3
>>> 7 % 2                   # 取余运算
1
>>> 5 ^ 3                   # 位异或，5 的二进制形式为 101，3 为 011，异或后为 110，即十进制的 6
6
>>> 5 ^ 5
0
>>> 11 | 5                  # 位或运算
15
>>> 12 & 12                 # 位与运算
12
>>> 2 * 5 ** 2              # 这里先计算 5 ** 2
50
>>> 2 + 3 * 5               # 这里先计算 3 * 5
17
>>> 2 + 3 ^ 5               # 这里先计算 2 + 3
0
>>> 2 + (3 ^ 5)             # 改变运算顺序，先计算 3 ^ 5=6
8
>>> 3 + 4 * 5 ** 2 -20      # 先计算 5 ** 2 = 25，然后计算 4 * 25 =100，再计算 3 + 100 = 103，
                            # 最后计算 103 - 20
```

```
83
>>> 4 >> 2                  # 右移两位相当于除以 4
1
>>> 4 >> 1                  # 右移一位相当于除以 2
2
>>> (2 + 3) * 5             # 改变运算顺序，先计算 2 + 3
25
```

3.2 Python 数据类型：字符串

Python 中的字符串用于表示和存储文本。字符串通常由单引号（'...'）、双引号（"..."）或者三引号（'''...'''，"""..."""）包围，其中由三引号包围的字符串可以由多行组成。在 Python 脚本中，大段的叙述性字符串通常由三引号包围。

3.2.1 Python 中的字符串

字符串中可以包含数字、字母、中文字符、特殊符号，以及一些不可见的控制字符，如换行符、制表符等。字符串的形式一般如下所示。

```
>>> str1 = 'abcd'              # 使用单引号
>>> str2 = "Python"            # 使用双引号
>>> str3 = '123'
>>> str4 = 'a = 1 + 2 ^ 3 * 4'
>>> str5 = 'Can\'t'            # 在字符串中使用转义字符包含一个单引号(下面介绍转义字符)
>>> str5
"Can't"
>>> str6 = "Can't"             # 使用双引号包含一个单引号
>>> str6
"Can't"
```

3.2.2 字符串中的转义字符

如上面例子中的变量 str5 所示，如果要在字符中包含控制字符，或者一些在 Python 中表示特殊含义的符号，需要使用转义字符。常见的转义字符如表 3-3 所示。

表 3-3 常见转义字符

转义字符	含义
\n	换行符
\t	制表符
\r	回车
\\	表示\
\'	表示一个单引号，而不是字符串结束
\"	表示一个双引号，而不是字符串结束

下面的代码演示了转义字符在字符串中的使用。

```
>>> t = 'Hi,\tPython!'         # 在字符串中加入制表符
```

```
>>> print(t)
Hi,        Python!              # 在 Hi,与 Python! 之间有一段制表符的距离
>>> t
'Hi,\tPython!'                  # 只有在使用 print 函数输出字符串时才会解释字符串中的转义字符
>>> t = 'Hi,\nPython!'          # 在字符串中加入换行符
>>> print(t)
Hi,
Python!
>>> t = 'Hi,\rPython!'          # 在字符串中加入回车，相当于使用换行符
>>> print(t)
Hi,
Python!
>>> t = 'Hi,\\nPython!'         # 如果在字符串中加入两个 "\"，则是对第 2 个 "\" 进行转义输入 "\"
字符本身
>>> print(t)
Hi,\nPython!
```

3.2.3 操作字符串

在 Python 中，可使用运算符 "+"、"*" 对字符串进行运算，另外，Python 中提供很多对字符串操作的函数，可对字符串进行操作。其中常用的对字符串操作函数如表 3-4 所示。

表 3-4 常用字符串操作函数

字符串操作	描述
string.capitalize()	将字符串的第一个字母大写
string.count()	获得字符串中某一子字符串的数目
string.find()	获得字符串中某一子字符串的起始位置
string.isalnum()	检测字符串是否仅包含 0-9A-Za-z
string.isalpha()	检测字符串是否仅包含 A-Za-z
string.isdigit()	检测字符串是否仅包含数字
string.islower()	检测字符串是否均为小写字母
string.isspace()	检测字符串中所有字符是否均为空白字符
string.istitle()	检测字符串中的单词是否为首字母大写
string.isupper()	检测字符串是否均为大写字母
string.join()	连接字符串
string.lower()	将字符串全部转换为小写
string.split()	分割字符串
string.swapcase()	将字符串中大写字母转换为小写、小写字母转换为大写
string.title()	将字符串中的单词首字母大写
string.upper()	将字符串中全部字母转换为大写
len(string)	获取字符串长度

函数的使用如下所示。

```
>>> str = 'hi, python!'          # 字符 "p" 之前有一个空格
>>> str.capitalize()             # 将字符串的第一个字母大写
```

```
'Hi, python!'
>>> str.count('p')                    # 获得字符串中 "p" 的数目
1
>>> str.find('hello')                 # 获得字符串中 "hello" 的起始位置
-1                                    #  -1 表示未找到
>>> str.find('p')                     # 获得字符串中 "p" 的起始位置
4                                     # 从 0 开始也就是字符串中第 5 个字符
>>> str.isalnum()                     # 检测字符串是否仅包含 0-9A-Za-z
False
>>> str.isalpha()                     # 检测字符串是否仅包含字母
False
>>> str.isdigit()                     # 检测字符串是否仅包含数字
False
>>> str.islower()                     # 检测字符串是否均为小写字母
True
>>> str.isspace()                     # 检测字符串中所有字符是否均为空白字符
False
>>> str.istitle()                     # 检测字符串中的单词是否为首字母大写
False
>>> str.isupper()                     # 检测字符串是否均为大写字母
False
>>> str.join('HI')                    # 连接字符串（以 str 字符串为分隔符连接参数中的每一项）
'Hhi, python!I'
>>> str.upper()                       # 将字符串全部转换为大写
'HI, PYTHON!'
>>> str.title()                       # 将字符串中的单词首字母大写
'Hi, Python!'
>>> str.split()                       # 以空格分割字符串
['hi,', 'python!']
>>> str.split(',')                    # 以 "," 分割字符串
['hi', ' python!']
>>> len(str)                          # 获取字符串长度
11
>>> str + 'hello'                     # 使用 "+" 连接字符串
'hi, python!hello'
>>> str * 3                           # 使用 "*" 重复字符串，此处重复三次
'hi, python!hi, python!hi, python!'
>>> str * 2                           # 此处重复两次
'hi, python!hi, python!'
>>>
>>> str                               # 输出 str
'hi, python!'                         # 仍为原来的字符串
```

在上述操作中，使用字符串操作函数，或使用运算符对字符串 str 进行操作时，并不会改变字符串 str 本身的内容，而是返回修改后的新字符串。如果要修改原字符串，则可以使用下面的代码。

```
>>> str = str.title()                 # 将 str.title()函数的返回值赋值给 str，即修改 str
>>> str
'Hi, Python!'
```

上述函数使用比较复杂的为 string.join()和 string.split()函数。string.join()函数将原字符串插入

参数字符串中的每两个字符之间。如果参数字符串中只有一个字符，则返回参数字符串。同样，string.join()并不改变原字符串，只是返回一个新的字符串。代码如下所示。

```
>>> str = 'how'                    # 原始字符串
>>> str.join('---')                # 将原始字符串插入 "---" 之中
'-how-how-'
>>> str.join('a')                  # 参数字符串只有一个字符
'a'                                # 返回参数字符串
>>> str                            # 经过操作后，原字符串并未改变
'how'
```

string.split()函数将字符串以指定的字符进行分割，如果不指定字符，则默认以空格分割字符串。其函数原型如下所示。

```
split( [sep [,maxsplit]])
```

其参数含义如下。

- sep　　可选参数，指定分割的字符。
- maxsplit　　可选参数，分割次数。

```
>>> str = 'Python is wonderful!'    # 定义原始字符串
>>> str.split()                     # 以空格分割字符串
['Python', 'is', 'wonderful!']      # 返回除去空格的字符串列表(分为了 3 个子串)
>>> str.split( None,1)              # 以空格分割，但只分割一次
['Python', 'is wonderful!']
>>> str.split( None,0)              # 相当于不分割
['Python is wonderful!']
>>> str.split( 'o',)                # 以字母 "o" 分割字符串
['Pyth', 'n is w', 'nderful!']
```

3.2.4 字符串的索引和分片

Python 中的字符串相当于一个不可变序列的列表。一旦声明一个字符串，则该字符串中的每个字符都有了自己固定的位置。在 Python 中可以使用 "[]" 来访问字符串中指定位置上的字符，这种方式类似于 C 语言中的数组。

与数组类似，在 Python 字符串中，字符的序号也是从 0 开始的，即 string[0]表示字符串 string 中的第一个字符。

与 C 语言不同的是，Python 还允许以负数表示字符的序号，负数表示从字符串尾部开始计算，此时最后一个字符的序号为-1，而不是-0。以下代码演示了如何使用 "[]" 访问字符串中的字符。

```
>>> str = 'abcdefg'          # 定义原始字符
>>> str[2]                   # 取字符串中序号为 2 的字符，也就是第 3 个字符
'c'
>>> str[-2]                  # 从字符串尾部开始计算，最后一个字符的序号为-1，str[-2]即取
                             # 倒数第 2 个字符
'f'
>>> str[-0]                  # -0 即 0，就是取字符串中第一个字符
```

```
'a'
>>> str[-1]                    # 取字符串中最后一个字符
'g'
>>> str[1:4]                   # 取从字符串中第 2 个字符到第 5 个字符的内容,但不包含第 5 个字符
'bcd'
>>> str[1:1]                   # 由于不包含第 2 个字符，故为空
''
>>> str[2:4]                   # 从第 3 个字符开始到第 5 个字符，但不包含第 5 个字符
'cd'
>>> str[1:-1]                  # 从第 2 个字符开始到最后一个字符，但不包含最后一个
'bcdef'
>>> str[0:-2]                  # 从第一个字符开始到倒数第 2 个字符，但不包含倒数第 2 个字符
'abcde'
>>> str[:-2]                   # 与 str[0:-2]相同
'abcde'
```

3.2.5 格式化字符串

在 Python 中，字符串中的字符顺序是不可变的。但是在某些情况下，可能又要根据不同的需要修改字符串的内容。这时，可使用 Python 的格式化字符串功能。

在 Python 中，可以在字符串中使用以“%”开头的字符，以在脚本中改变字符串中的内容。常用的格式化字符如下。

- %c　　单个字符。
- %d　　十进制整数。
- %o　　八进制整数。
- %s　　字符串。
- %x　　十六进制整数，其中的字母小写。
- %X　　十六进制整数，其中的字母大写。

以下代码演示了在字符串中使用格式化字符的方法。

```
>>> s = 'So %s day!'                # 定义字符串，在字符串中使用%s
>>> print(s % 'beautiful')          # 使用 beautiful 替换%s
So beautiful day!
>>> s % 'beautiful'                 # 与 print(s % 'beautiful')功能相同
'So beautiful day!'
>>> '1 %c 1 %c %d' % ('+','=',2)    # 使用多个格式化字符
'1 + 1 = 2'
>>> 'x = %x' % 0xA                  # 使用%x 格式化十六进制数字，其中的字母小写
'x = a'
>>> 'x = %X' % 0xa                  # 使用%X 格式化十六进制数字，其中的字母大写
'x = A'
```

3.2.6 字符串、数字类型的转换

在某些情况下，字符串可能完全由数字组成，并且该字符串还需要在脚本中进行算术运算。此时最简单的操作就是使用 int()函数将字符串转换为数字，将数字转换为字符串可以使用 str()函数。

以下代码演示了在 Python 中字符串与数字的相互转换。

```
>>> '10' + 4                    # 两种不同类型对象相加引发异常
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> int('10') + 4               # 将字符串转换为数字
14
>>> '10' + str( 4 )             # 将数字转换为字符串
'104'
```

在 Python 2.x 中，很多程序员习惯使用 string 模块中的 atoi()函数将字符串转换为整数。其函数的原型如下。

```
string.atoi( s[, base])
```

其参数的含义如下。

- s　　进行转换的字符串。
- base　　可选参数，表示将字符转换成指定进制的数字。

以下代码演示了在 Python 2.x 中用 string.atoi 将字符串转换为整数的方法。

```
>>> string.atoi('10') + 4        # 将字符串转换为数字
14
```

不过，这种方法在 Python 3 中已经不能使用了。在 Python 3 的 locale 模块中提供了 atoi()函数，可将字符串转换为整数。locale 模块提供了 C 语言本地化（localization）函数的接口，同时提供相关函数，实现基于当前 locale 设置的数字、字符串转换。

3.2.7 原始字符串（Raw String）

原始字符串是 Python 中一类比较特殊的字符串，以大写字母 R 或者小写字母 r 开始。在原始字符串中，字符“\”不再表示转义字符的含义。

原始字符串是为正则表达式设计的，也可以用来方便地表示 Windows 系统下的路径，不过，如果路径以“\”结尾那么会出错。下面是演示代码。

```
>>> import os
>>> path = r'e:\book'            # 使用原始字符串
>>> os.listdir(path)             # 列出目录中的内容
['res', 'code', 'bak', 'temp']
>>> os.listdir('e:\book')        # Python 2.x 执行这个命令会报错，Python 3 中可正常执行
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
WindowsError: [Error 22] : 'e:\x08ook/*.*'
>>> path = R'e:\book\'           # 原始字符串中不能以“\”结尾
Traceback (  File "<interactive input>", line 1
   path = R'e:\book\'
                   ^
SyntaxError: EOL while scanning string literal
```

3.3 Python 数据类型：列表和元组

Python 提供了两个可保存大量数据的数据类型：列表和元组。本节将介绍这两种数据类型的特点和基本操作。

3.3.1 创建和操作列表

列表是以方括号“[]”包围的数据集合，不同成员间以“,”（半角符号）分隔。列表中可以包含任何数据类型，也可包含另一个列表，列表可以通过序号来访问其中的成员。在脚本中可以对列表进行排序、添加、删除等操作，以改变列表中某一成员的值。Python 提供了对列表操作的强大支持，常用的操作如表 3-5 所示。

表 3-5 常用列表操作

列表操作	描述
list.append()	追加成员
list.count(x)	计算列表中的参数 x 出现的次数
list.extend(L)	向列表中追加另一个列表 L
list.index(x)	获得参数 x 在列表中的位置
list.insert()	向列表中插入数据
list.pop()	删除列表中的最后一个值，并返回被删除的值
list.remove()	删除列表中的成员
list.reverse()	将列表中成员的顺序颠倒
list.sort()	对列表中的成员排序

在 Python 中，除了可以使用表 3-5 所示的函数操作列表之外，还可以使用类似于字符串的分片和索引操作列表。

以下代码演示了在 Python 中列表的使用。

```
>>> list = []                          # 定义一个空列表
>>> list.append( 1 )                   # 向列表中添加成员
>>> list.count( 2 )                    # 计算 2 在列表中出现的次数
0
>>> list.extend( [ 2, 3 , 5, 4] )  # 向列表中添加一个列表
>>> list
[1, 2, 3, 5, 4]                        # 列表值被改变
>>> list.index( 5 )                    # 获得 5 在列表中的位置
3                                      # 从 0 开始，即第 4 个
>>> list.insert( 2, 6)                 # 从 0 开始，也就是在第 3 个成员处插入 6，其他成员依次后移
>>> list
[1, 2, 6, 3, 5, 4]
>>> list.pop(2)                        # 删除列表中第 3 个成员
6
>>> list
[1, 2, 3, 5, 4]
>>> list.remove(5)                     # 删除列表中的 5
```

```
>>> list
[1, 2, 3, 4]
>>> list.reverse()                    # 颠倒列表的顺序
>>> list
[4, 3, 2, 1]
>>> list.sort()                       # 将列表中成员重新排序
>>> list
[1, 2, 3, 4]
```

3.3.2 创建和操作元组

元组的特性与列表基本相同，元组是以圆括号“()”包围的数据集合。与列表不同的是，元组中的数据一旦确立就不能被改变。元组可以使用在不希望数据被其他操作改变的场合。

而对于元组，由于其内容不能被改变，因此不能使用表 3-5 所示的函数进行操作，只能对其使用分片和索引操作。

以下代码演示了在 Python 中元组的使用（接着上面对列表的操作，在下面命令中将使用列表 list）。

```
>>> tuple = ( 'a', 'b', 'c')     # 定义一个元组
>>> list.insert(4,tuple)     # 向列表中插入一个元组
>>> list
[1, 2, 3, 4, ('a', 'b', 'c')]
>>> list[4]                  # 使用索引访问列表中的第 5 个成员
('a', 'b', 'c')
>>> list[1:4]                # 使用分片获得列表中第 2 个至第 5 个成员，但不包含第 5 个成员
[2, 3, 4]
>>> tuple[2]                 # 获得元组中第 3 个成员
'c'
>>> tuple[1:-1]              # 获得元组中第二个程序至最后一个程序，但不包含最后一个程序
('b',)
```

3.4 Python 数据类型：字典

字典是 Python 中比较特别的一类数据类型，是以大括号包围“{}”的数据集合。字典与列表的最大不同在于字典是无序的，在字典中通过键来访问成员。

与列表类似，字典也是可变的，可以包含任何其他类型，字典中的成员位置只是象征性的，并不能通过其位置来访问该成员。字典中的成员是以“键：值”的形式来声明的。常用的字典操作如表 3-6 所示。

表 3-6 常用字典操作

字典操作	描述
dic.clear()	清空字典
dic.copy()	复制字典
dic.get(k)	获得键 k 的值
dic.has_key(k)	是否包含键 k
dic.items()	获得由键和值组成的列表

续表

字典操作	描述
dic.keys()	获得键的列表
dic.pop(k)	删除键 k
dic.update()	更新成员
dic.values()	获得值的列表

以下代码演示了 Python 中字典的基本操作。

```
>>> dic = { 'apple':2, 'orange':1 }     # 定义一个字典
>>> dic.copy()                          # 复制字典
{'orange': 1, 'apple': 2}
>>> dic['banana'] = 5                   # 增加一项
>>> dic.items()
dict_items([('orange', 1), ('apple', 2), ('banana', 5)])   # 获得字典中成员的列表
>>> dic.pop('apple')                    # 删除 "apple"，并返回其值
2
>>> dic
{'orange': 1, 'banana': 5}
>>> dic.pop('apple',3)                  # 删除 "apple"，如果没有 "apple" 则返回 3
3
>>> dic.keys()                          # 获得键的列表
dict_keys(['orange', 'banana'])
>>> dic.values()                        # 获得值的列表
dict_values([1, 5])
>>> dic.update({'banana':3})            # 更新 "banana" 的值
>>> dic
{'orange': 1, 'banana': 3}
>>> dic.update({'apple':2})             # 更新 "apple" 的值，如果没有则添加
>>> dic
{'orange': 1, 'apple': 2, 'banana': 3}
>>> dic['orange']                       # 通过键获取值
1
>>> dic.clear()                         # 清空字典
>>> dic
{}
```

3.5 Python 数据类型：文件

文件也可以看作是 Python 中的数据类型。当使用 Python 的内置函数 open 打开一个文件后就返回一个文件对象。其原型如下。

```
open(filename, mode, bufsize)
```

其参数含义如下。

- filename　　要打开的文件名。
- mode　　可选参数，文件打开模式。

◆ bufsize　　可选参数，缓冲区大小。

其中 mode 可以是 "r"，表示以读方式打开文件，也可以是 "w"，表示以写方式打开文件，"b" 表示以二进制方式打开文件。常用的文件操作如表 3-7 所示。

表 3-7　常用文件操作

文件操作	描述
file.read()	将整个文件读入字符串中
file.readline()	读入文件的一行到字符串中
file.readlines()	将整个文件按行读入到列表中
file.write()	向文件中写入字符串
file.wirtelines()	向文件中写入一个列表
file.close()	关闭打开的文件

以下代码演示了 Python 中文件的基本操作。

```
>>> file = open('c:/python.txt','w')    # 打开 C 盘下的 python.txt 文件，如果没有则创建
>>> file.write('python\n')              # 向文件中写入字符
7
>>> a = []                              # 定义空列表
>>> for i in range(10):                 # 循环向列表中添加字符
...       s = str(i) + '\n'
...       a.append(s)
...
>>> file.writelines(a)                  # 将列表写入文件
>>> file.close()                        # 关闭文件
>>> file = open('c:/python.txt','r')    # 重新以读方式打开文件
>>> s = file.read()                     # 读取整个文件
>>> print(s)                            # 输出文件内容
python
0
1
2
3
4
5
6
7
8
9
# 关闭文件，然后使用 readlines 读取文件
# 如果不关闭文件，则读取的内容为空
# 因为文件内容已经被读入到变量 s 中
>>> file.close()
>>> file = open('c:/python.txt','r')
>>> l = file.readlines()                # 将文件读取到列表中
>>> print(l)                            # 输出列表
['python\n', '0\n', '1\n', '2\n', '3\n', '4\n', '5\n', '6\n', '7\n', '8\n', '9\n']
```

3.6 Python 的流程控制语句

通常情况下，Python 脚本总是按顺序执行的。不过，对于一些复杂的程序，可能需要根据执行过程中出现的不同情况选择性地执行一部分语句而跳过另一部分语句，或重复执行某一部分语句，这时，就需要使用 Python 的流程控制语句。

Python 脚本中的流程控制语句控制着脚本的执行流程，根据一定的条件来执行脚本中不同的语句，以完成不同的功能。

3.6.1 分支结构：if 语句

if 语句是基本的条件测试语句，用来判断可能遇到的不同情况，并针对不同的情况选择执行某一部分语句。if 语句的基本形式如下，其在脚本中的执行流程如图 3-1 所示。

```
if <条件>:              # 当条件为真时，执行缩进的语句；当条件为假时，判断 elif 的条件
   <语句>               # 用缩进来表示语句处于 if 语句之中
elif <条件>:            # 当条件为真时，执行缩进的语句，当条件为假时，执行 else 语句
   <语句>
else:                  # 若前边所有的条件都为假，则执行下面的缩进语句
   <语句>
```



图 3-1 if 语句执行流程

在条件语句中，主要使用如表 3-8 所示的几种比较运算符。

表 3-8 比较运算符

比较运算符	含义
a == b	a 与 b 是否相等，是则返回真，否则返回假

续表

比较运算符	含义
a != b	a 是否不等于 b，是则返回真，否则返回假
a > b	a 是否大于 b，是则返回真，否则返回假
a >= b	a 是否大于等于 b，是则返回真，否则返回假
a < b	a 是否小于 b，是则返回真，否则返回假
a <= b	a 是否小于等于 b，是则返回真，否则返回假

以上几种比较运算符可以用于对数字、字符串、列表、元组以及字典等的比较。除了上述的比较运算符外，在条件中也可以使用逻辑运算及一些其他的语句。以下代码演示了 Python 中基本的 if 条件测试语句。

```
>>> a = 1
>>> b = 2
>>> if a == b:                    # 判断 a 与 b 是否相等
...     print('true')             # 相等则打印 true
... else:
...     print('false')            # 否则打印 false
...                                # 此处要多按一下回车键，脚本才会执行 if 语句
false
>>> if a < b:                     # 判断 a 是否小于 b
...     print('true')             # 如果 a 小于 b 则打印 true
... else:
...     print('false')            # 否则打印 false
...
true
>>> m = 'hi'
>>> n = 'hello'
>>> if m == n:                    # 此处为字符串比较
...     print('true')
... elif m > n:
...     print('false')
... else:
...     print(m,n)
...
false
>>> l1 = [1,2]
>>> l2 = [3,4]
>>> if l1 == l2:                  # 此处为列表比较，只包含 if 语句块
...     print('true')
...
>>> if l1 != l2:
...     print('false')
...
false
>>> if l1 <= l2:
...     print('true')
...
true
>>> if not 1:                     # 逻辑运算非，相当于 if 0:，即条件为假
...     print('true')
... else: print('false')
```

```
...
false
```

在 if 语句中还可以嵌套其他的 if 语句，被包含的 if 语句要用缩进来表示自己所包含的语句，这是 Python 独特的语法，而不像其他语言那样使用一对大括号 "{}" 来表示一个语句块。虽然缩进可以使脚本看起来更加清晰，但在编写的过程中容易忽略缩进，而导致程序语法错误，或者导致结果出错。在编写 Python 脚本中最好使用具有自动缩进功能的编辑器，以保证程序正确缩进，减少敲击键盘的次数。在 if 语句中嵌套其他 if 语句的结构如下所示。

```
if <条件>:
    if <条件>:              # 嵌入的 if 语句
        <语句>              # 此处相对于 if 再缩进
    else:
        <语句>              # 此处相对于 else 再缩进
elif<条件>:
    if<条件>:
        <语句>
    <其他语句>              # 此语句未缩进，表示不属于嵌入的 if
else:
    <语句>
```

3.6.2 循环结构：for 语句

for 语句是 Python 中的循环控制语句。for 语句可以用于循环遍历某一对象，它还具有一个附带的 else 块。附带的 else 块是可选的，主要用于处理 for 语句中包含的 break 语句。for 语句中的 break 语句，可以在需要的时候终止 for 循环。如果 for 循环未被 break 语句终止，则会执行 else 块中的语句。在 for 语句中还可以使用 continue 语句，continue 语句可以跳过位于其后的语句，开始下一轮循环。for 语句的格式如下。

```
for <> in <对象集合>:
    if<条件>:
        break               # 终止循环
    if<条件>:
        continue            # 使用 continue 跳过其他语句，继续循环
    <其他语句>
else:                       # 如果 for 循环未被 break 语句终止，则执行 else 块中的语句
    <>
```

以下是一个完整的 for 循环语句。

```
>>> for i in [ 1,2,3,4,5 ]:
... if i == 6:
...     break
... if i == 2:
...     continue
... print(i)
... else:
... print('all')
...
```

```
1
3
4
5
all
>>>
```

for 语句中的对象集合可以是列表、字典或元组等。也可以通过 range()函数产生一个整数列表，以完成计数循环。range()函数的原型如下。

```
range( [start,] stop[, step])
```

其参数含义如下。

- start　　可选参数，起始数。
- stop　　终止数，如果 range 只有一个参数 x，那么 range 生产一个从 0 至 x—1 的整数列表。
- step　　可选参数，步长。

以下代码使用 for 和 range 函数输出 1~5。

```
>>> for i in range(1, 5 + 1):        # 用 range 函数产生一个包含 1 到 5 的整数列表
... print(i)
...
1
2
3
4
5
```

以下代码是使用 for 语句遍历一个字典。由于 for 语句遍历的是字典的键，这样就可以使用 dic[key]的形式同样遍历字典中的值。

```
>>> people = {'Tom':170, 'Jack':175, 'Kite':160, 'White':180}
>>> for name in people:
... print(people[name])
...
180
160
175
170
```

在 for 循环中，除了循环的对象可以是元组以外，循环的目标也可以是元组，可以在循环的过程中对元组进行赋值等操作。以下代码是在 for 循环中使用元组。

```
>>> tt = ( ('a','b'),('c','d'),('e','f'),('g','h') )
>>> for t1 in tt:                              # 此处的 t1 相对于一个元组
... print(t1)
...
('a', 'b')
('c', 'd')
('e', 'f')
('g', 'h')
>>> for (x,y) in tt:                           # 循环的目标为一个元组
```

```
... print(x,y)
...
a b
c d
e f
g h
```

以下是一个比较复杂的 for 循环，通过在 for 循环中嵌套 for 循环语句及 if 语句，实现求解 50 至 100 之间的全部素数。

```
>>> import math                                  # 导入 math 模块，已使用求平方根的函数
>>> for i in range(50,100 + 1):                  # 遍历 50~100
... for t in range( 2, int(math.sqrt(i)) + 1 ):     # 从 2 到 i 的平方根，此处使用 int
将 i 的平方根转为整数
...     if i % t == 0:                            # 判断 i 能否被 2 到 i 的平方根内的数整除
...         break                                 # 终止循环，即 i 不是素数
... else:
...     print(i)                                  # 如果循环没有被 break 终止，即 i 为素数，打印 i
...
53
59
61
67
71
73
79
83
89
97
```

3.6.3 循环结构：while 语句

while 语句也是循环控制语句，与 for 循环不同的是，while 语句只有在测试条件为假时才会停止。在 while 的语句块中，一定要包含改变测试条件的语句，以保证循环能够结束，避免死循环的出现。

while 语句包含与 if 语句相同的条件测试语句，如果条件为假，则终止循环。while 语句有一个可选的 else 语句块，与 for 循环中的 else 语句块一样，当 while 循环不是由 break 语句终止时，则会执行 else 语句块中的语句。continue 语句也可以用于 while 循环中，其作用与 if 语句中的 continue 相同，都是跳过 continue 后的语句，进入下一个循环。while 的一般格式如下。

```
while <条件>:
    if <条件>:
        break           # 终止循环
    if <条件>:
        continue        # 跳过后面的语句
    <其他语句>
else:
    <语句>              # 如果循环未被 break 语句终止，则执行
```

while 循环不像 for 循环可以遍历某一对象的集合，while 循环最容易出现的问题就是测试条件

永远为真，导致死循环。因此在使用 while 循环时，应仔细检查 while 语句的测试条件，避免出现死循环。

以下代码是使用 while 语句打印数字 1～5。

```
>>> x = 1
>>> while x <= 5:          # 条件测试
... print(x)
... x = x + 1              # 改变条件，使 x 增加，以结束循环
...
1
2
3
4
5
```

以下代码是使用 while 语句访问一个列表。从代码可以看出，使用 while 语句遍历列表要比使用 for 语句遍历列表复杂一些。

```
>>> l = [ 'a', 'b', 'c', 'd', 'e' ]
>>> i = len( l )           # 获得列表长度
>>> while i != 0:          # 条件测试
... print(l[-i])           # 因为 i 从大到小递减，故使用负值，从头输出列表的值
... i = i - 1              # 条件控制语句，避免死循环
...
a
b
c
d
e
```

3.7 本章小结

通过对本章的学习，可以知道 Python 提供了丰富的数据类型。除了其他程序设计语言也提供的数字和字符串之外，Python 还提供了列表、元组、字典、文件等多种内置数据类型。对于一般的 Python 脚本来说，使用 Python 的内置数据类型可以完成绝大多数工作，基本不必考虑重新自定义数据类型或是数据结构。而在程序结构控制方面，Python 提供了三种基本的语句，if 分支、for 循环、while 循环，用这三种流程控制语句已经可以很好地完成程序流程控制。

在下一章，将介绍 Python 的函数与模块，学习通过函数和模块来封装脚本。

◆ ◆ ◆

第 4 章　可复用的函数与模块

本章包括

- Python 自定义函数
- 变量的作用域
- Python 模块
- 函数参数的使用
- 用 lambda 声明函数
- 用包来管理多个模块

函数是一组语句的集合，用以实现某一特定的功能。函数可以简化脚本，Python 本身提供了许多内置函数，极大地方便了脚本的编写。例如，可以使用 print 函数输出计算结果，使用 input 函数接收用户的输入。除了系统内置的函数之外，程序员还可以根据需要编写自己的函数。

当自定义函数很多时，为了方便这些函数的管理，可将函数分类保存到不同的模块中。因此，模块可以看作是一组函数的集合。很多函数库在 Python 中都是以模块的形式提供的，例如，系统中的 os 模块提供了对系统操作的一系列函数。

4.1 Python 自定义函数

在编写脚本的过程中，经常要完成许多重复的工作。此时就可以将完成重复工作的语句提取出来，将其编写为函数。在脚本中可以方便地调用函数来完成这些重复的工作，而不必重复的复制粘贴代码。在前面的章节中已经介绍了 Python 内置的部分函数。

在 Python 中，函数必须先声明，然后才能在脚本中使用。使用函数时，只要按照函数定义的形式向函数传递必需的参数，就可以调用函数完成所需的功能。

4.1.1 函数声明

在 Python 中，使用 def 可以声明一个函数。完整的函数是由函数名、参数以及函数实现语句组成的。同前面几章中所讲解的 Python 基本语句一样，在函数中也要使用缩进来表示语句属于函数体。

如果函数有返回值，那么需要在函数中使用 return 语句返回计算结果。声明函数的一般形式如下。

```
def <函数名> (参数列表):
    <函数语句>
    return <返回值>
```

其中参数和返回值不是必须的。很多函数可能既不需要传递参数，又没有返回值。例如，下面定义的一个简单函数。

```
# hi 是所声明函数的函数名，以便在脚本中使用该函数
```

```
# 虽然不需要传递参数，但在函数声明的时候依然要在函数名后跟一对圆括号
>>> def hi ():
...     print('hi,python!')      # 缩进的语句，表示是函数内的语句
...                              # 函数没有使用 return 定义返回值
```

以下是一个完整的函数，实现了求一个列表中所有整数之和的功能。其参数 L 为所要求和的列表，result 是列表中所有整数的和，最后函数使用 return 将 result 返回。函数声明代码如下。

```
>>> def ListSum( L ):
...     result = 0
...     for i in L:
...         result = result + i
...     return result
```

Python 的函数比较灵活。与 C 语言中函数的声明相比，在 Python 中声明一个函数时，不需要声明函数类型，也不需要声明参数的类型。Python 在实际调用函数的过程中也非常灵活，不需要为不同类型的参数声明多个函数，或在处理不同类型数据的时候调用相应的函数。大部分情况下都可以用同一个函数调用不同的数据类型。

如下所示的函数，其功能是打印参数对象中的所有成员。

```
>>> def PrintAll ( X ):      # 声明函数
... for x in X:              # 遍历参数
...     print(x)
...
>>> l = [ 1, 2, 3, 5]        # 定义一个列表
>>> PrintAll(l)              # 打印列表中的内容
1
2
3
5
>>> t = ( 'a','b','c')       # 定义一个元组
>>> PrintAll(t)              # 打印元组中的内容
a
b
c
```

上述代码只声明了一个 PrintAll(X)函数，并没有指定参数的类型。函数调用的时候，不仅可以向其传递一个列表，还可以向其传递一个元组。可以看到，不管参数为一个列表，还是一个元组，函数都能被正确地执行。

当然，这并不表示可以向函数传递任何参数，其主要还是取决于函数的实现。在 PrintAll(x)函数中，只使用了 for 循环语句，以及 print 函数，它们都对所操作的对象没有特别的要求，因此函数才得以正确地执行。

虽然 Python 中的函数灵活性很强，但这也意味着一旦出现问题只有在脚本运行的时候才能被发现。

4.1.2 函数调用

在以 PrintAll(x)函数为例时，已经演示了如何调用函数。在 Python 中调用自定义函数与系统内

置函数的方法相同，只要使用函数名指定要调用的函数，然后在函数名后的圆括号中给出函数的参数即可。如果有多个参数，则不同的参数要以“,”隔开。

需要注意的是，即使函数不需要参数，也要在参数名后使用一对空的圆括号。

函数调用必须在函数声明之后。

再来看函数调用的代码，如下所示。

```
>>> def hi ():                  # 上一小节中定义的函数，不需要传递参数，也没有返回值
... print('hi,python!')
...
>>> hi()                        # 调用函数，使用一对空括号
hi,python!                      # 函数运行的结果，而不是返回值
>>> hi                          # 只输入函数名，而不加括号
<function hi at 0x01124770># 返回的是函数在内存中地址
>>> def ListSum( L ):           # 上一小节中定义的函数，求解列表中的整数之和
... result = 0
... for i in L:
...     result = result + i
... return result
>>> l = [ 1, 2, 3, 4, 5]        # 定义一个列表
>>> r = ListSum( l )            # 调用 ListSum 函数，传递 l 为参数，将 r 赋值为函数的返回值
>>> print(r)                    # 输出 r 的值
15
>>> r = hi()                    # 将 r 赋值为 hi 函数的返回值
hi,python!
>>> print(r)                    # 输出 r 的值
None                            # 表示函数无返回值，也可以理解为函数返回的值为 None
```

4.2 参数让函数更有价值

在 Python 中，函数的参数除了 4.1 节中介绍的一种方式之外，还有很多种形式。例如，在调用某些函数时，既可以向其传递参数，也可以不传递参数，函数依然可以正确调用。还有一些情况，当函数中的参数数量不确定，可能是 1 个，也可能是几个、甚至几十个时，对于这些函数，应该如何定义其参数呢?

4.2.1 有默认值的参数

在 Python 中，可以在声明函数时，预先为参数设置一个默认值。当调用函数时，如果某个参数具有默认值，则可以不向函数传递该参数，这时，函数将使用声明函数时为该参数设置的默认值。

声明一个参数具有默认值的函数形式如下。

```
def <函数名> (参数=默认值):
    <语句>
```

以下代码声明了一个函数，用来计算参数的立方值，其参数的默认值为 5。

```
>>> def Cube ( x = 5 ):# 声明函数，将参数默认值设置为 5
```

```
... return x ** 3
...
>>> Cube(2)                    # 调用函数，计算 2 的立方
8
>>> Cube()                     # 调用函数，计算默认参数的立方
125
```

如果一个函数具有多个参数，并且这些参数都具有默认值，在调用函数的时候，可能仅想向最后一个参数传递值，该怎么办呢？先来看下面的代码。

```
>>> def Cube( x = 1, y = 2, z= 3 ):      # 声明函数
... return ( x + y - z) ** 3
...
>>> Cube ( 0 )                           # 向参数传递一个值，是传递给 x
-1
>>> Cube ( 3, 3)                         # 向参数传递两个值，是传递给 x，y
27
>>> Cube ( , ,5 )                        # 这样是错误的
Traceback (  File "<interactive input>", line 1
   Cube ( , ,5 )
          ^
SyntaxError: invalid syntax
```

从上述代码可以看出，在 Python 中，传递参数是按照声明函数时定义的参数的顺序依次传递的。如果在调用函数时使用且仅使用 "," 表示向函数的最后一个参数传递值的话，则会引发错误。如果需要向指定的参数传递值，则可以使用以下方式重新定义一下函数。

```
>>> def Cube( x = None, y = None, z= None ):   # 声明函数，将参数默认值均设为 None
... if x == None:                              # 判断参数 x 的值是否为 None，即未传递值
...     x = 1                                  # 将 x 赋值为 1，相当于参数 x 的默认值为 1
... if y == None:                              # 判断参数 y 的值是否为 None，即未传递值
...     y = 2                                  # 将 y 赋值为 2，相当于参数 y 的默认值为 2
... if z == None:                              # 判断参数 z 的值是否为 None，即未传递值
...     z = 3                                  # 将 z 赋值为 3，相当于参数 x 的默认值为 3
... return ( x + y - z ) ** 3
...
>>> Cube ( )                                   # 调用函数，使用参数的默认值
0
>>> Cube( None, None, 5)                       # 调用函数，仅 x，y 使用默认值
-8
```

除了上述方法外，还可以使用下一小节中的方法，向指定的参数传递值。

4.2.2 参数的传递方式

在 Python 中，参数值的传递不只是按照声明函数时参数的顺序进行传递的，实际上，Python 还提供了另外一种传递参数的方法——按照参数名传递值。以参数名传递参数时类似于设置参数的默认值。

使用按参数名传递参数的方式调用函数时，要在调用函数名后的圆括号里为函数的所有参数赋值，赋值的顺序不必按照函数声明时的参数顺序，代码如下。

```
>>> def fun ( x, y, z):                    # 声明函数
... return x + y - z
...
>>> fun ( 1, 2, 3)                         # 调用函数，按顺序传递参数
0
>>> fun ( z = 1, x = 2, y = 3)             # 调用函数，按照参数名传递参数
4
```

在 Python 中，调用函数可以同时使用按顺序传递参数和按参数名传递参数两种方式。但是，需要注意的是，按顺序传递的参数要位于按参数名传递的参数之前，而且不能有重复的情况。代码如下。

```
>>> def mysum( x, y, z):                   # 声明函数
... return x + y + z
...
>>> mysum ( 1, z = 3, y = 2)               # 同时使用按顺序传递参数和按参数名传递参数
6
>>> mysum( z = 3,y =2, 1)   # 错误的方式！按顺序传递的参数不能位于按参数名传递的参数之后
Traceback (  File "<interactive input>", line 1
SyntaxError: non-keyword arg after keyword arg (<interactive input>, line 1)
>>> mysum ( 5, z = 6, x = 7)    # 错误的方式！参数重复，5 已经传递给 x，后边又将 7 传递给 x
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: mysum() got multiple values for keyword argument 'x'
```

在具有默认参数值的函数中，使用按参数名传递参数的方法非常方便。例如，4.2.1 节的例子中，如果使用按照参数名向函数传递参数的方法，就不必在函数声明时将参数的默认值设置为 None，可以省去函数中的判断语句。不过，为了让函数更具通用性，以下代码依然将参数的默认值设置为 None。具体如下。

```
>>> def Cube( x = None, y = None, z= None ):   # 声明函数
... if x == None:
...     x = 1
... if y == None:
...     y = 2
... if z == None:
...     z = 3
... return ( x + y - z ) ** 3
...
>>> Cube(z = 5)                              # 按参数名向函数中的参数 z 传递值
-8
>>> Cube( y = 6, z = 3)                      # 按参数名向函数中的参数 y，z 传递值
64
```

4.2.3 如何传递任意数量的参数

在 Python 中，函数可以具有任意个参数，而不必在声明函数时对所有参数进行定义。使用可变长参数的函数时，其所有参数都保存在一个元组里，在函数中可以使用 for 循环来处理。声明函数时，如果在参数名前加上一个星号 "*"，则表示该参数是一个可变长参数。示例代码如下。

```
>>> def mylistappend( *list ):          # 声明一个可变长参数的函数
...     l = []
...     for i in list:                   # 循环处理参数
...         l.extend( i )                # 将所有参数中的列表合并到一起
...     return l
...
>>> a = [ 1, 2, 3 ]                     # 定义列表
>>> b = [ 4, 5, 6 ]
>>> c = [ 7, 8, 9 ]
>>> mylistappend( a, b )                # 调用函数，传递两个参数
[1, 2, 3, 4, 5, 6]
>>> mylistappend( a, b, c )             # 调用函数，传递三个参数
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4.2.4 用参数返回计算结果

在 C 语言中，可以通过在参数中使用指针来达到改变参数值的作用，从而达到从函数中返回结果的目的。其实，在 Python 中，还有更简单的实现方法，即在参数中使用可变对象（如列表等），使函数中的结果返回到参数中，从而达到从函数中返回计算结果的目的。代码如下。

```
>>> def ChangeValue1( x ):          # 此处参数 x 应为整数
...     x = x ** 2
...
>>> def ChangeValue2( x ):          # 此处参数 x 应为列表
...     x[0] = x[0] ** 2
...
>>> a = 2                           # a 为整数，其值为 2
>>> b = [2]                         # b 为列表，其第一个成员为 2
>>> ChangeValue1(a)                 # 使用函数改变 a 的值，但不成功
>>> a
2
>>> ChangeValue2(b)                 # 使用函数改变 b 成员的值
>>> b
[4]                                 # 值被成功改变
```

4.3 变量的作用域

在 Python 脚本中，不同的函数可以具有相同的参数名。在函数中已经声明过变量名，在函数外还可以继续使用。而在脚本运行的过程中，其值并不相互影响。代码如下。

```
>>> def fun1( x ):                  # 声明一个函数
...     a = [ 1 ]                   # 定义一个名为 a 的列表
...     a.append(x)
...     print(a)
...
>>> a = [ 2, 3, 4 ]                 # 在函数外定义一个名为 a 的列表
>>> fun1(2)                         # 调用函数，输出函数中列表的值
[1, 2]
>>> a                               # 输出函数中名为 a 的列表值
```

```
[2, 3, 4]                          # 两者值并不相同
```

上述代码中，两个同名的列表之所以值不同，是因为它们处于不同的作用域里。在 Python 中，作用域可以分为内置作用域、全局作用域和局部作用域。内置作用域是 Python 预先定义的，全局作用域是所编写的整个脚本，局部作用域是某个函数内部范围。

上述代码中，函数中的列表 a 处于局部作用域中。而函数外的列表 a 处于全局作用域内。局部作用域内变量的改变并不影响全局作用域内的变量，除非通过引用的形式传递参数。

如果要在函数中使用函数外的变量，则可以在变量名前使用 global 关键字。代码如下。

```
>>> def fun( x ):                  # 声明函数
... global a                       # 使用 global 关键字声明全局变量
... return a + x
...
>>> a = 5                          # a 为全局变量，即 fun 函数中的 a
>>> fun(3)                         # 调用函数
8
>>> a = 2                          # 修改 a 的值
>>> fun(3)                         # 再次调用函数
5                                  # 返回值改变
```

4.4 最简单的函数：用 lambda 声明函数

用 lambda 表达式来声明函数，是 Python 中一类比较特殊的声明函数的方式，lambda 来源于 LISP 语言。使用 lambda 可以声明一个匿名函数。所谓匿名函数是指所声明的函数没有函数名，lambda 表达式就是一个简单的函数。使用 lambda 声明的函数可以返回一个值，在调用函数时，直接使用 lambda 表达式的返回值。使用 lambda 声明函数的一般形式如下。

```
lambda 参数列表:表达式
```

以下代码使用 lambda 定义了一个函数，并调用这个函数。

```
>>> fun = lambda x: x * x - x          # 使用 lambda 定义一个函数，返回函数地址
>>> fun(3)                             # 调用 lambda 定义的函数
6                                      # 函数返回值
>>> fun                                # fun 实际指向 lambda 定义的函数地址
<function <lambda> at 0x0111B970>
```

lambda 适用于定义小型函数。与 def 声明的函数不同，使用 lambda 声明的函数，在函数中仅能包含单一的参数表达式，而不能包含其他的语句。在 lambda 中也可以调用其他的函数。代码如下。

```
>>> def show():                        # 使用 def 声明 show 函数
... print('lambda')
...
>>> f = lambda:show()                  # 在 lambda 中调用 show 函数
>>> f()                                # 调用使用 lambda 生成的函数
```

```
lambda
>>> def shown( n ):                          # 使用 def 声明 shown 函数
... print('lambda' * n)
...
>>> fn = lambda x : shown( x )               # 在 lambda 中向 shown 函数传递值
>>> fn(2)                                    # 调用 lambda 生成的函数
lambdalambda
>>> def usereturn( x ):                      # 使用 def 声明 usereturn 函数
... return x*2
...
>>> fr = lambda x:usereturn(x) * x           # 在 lambda 函数中使用 usereturn 函数的返回值
>>> fr(3)
18
>>> fun = lambda x: print(x)                 # 不能在 lambda 中使用 print 语句
Traceback (  File "<interactive input>", line 1
   fun = lambda x: print x
                   ^
SyntaxError: invalid syntax
>>> fun = lambda x: if x <= 0 : x = -x # 不能在 lambda 表达式中使用其他语句
Traceback (  File "<interactive input>", line 1
   fun = lambda x: if x <= 0 : x = -x
                   ^
SyntaxError: invalid syntax
```

4.5 可重用结构：Python 模块

Python 中的模块实际上就是包含函数或者类的 Python 脚本。对于一个大型的脚本而言，经常需要将其功能细化，将实现不同功能的代码放在不同的脚本中实现，在其他的脚本中以模块的形式使用细化的功能，以便于脚本的维护和重用。本节就来介绍 Python 模块的相关内容。

4.5.1 Python 模块的基本用法

模块是包含函数和其他语句的 Python 脚本文件，它以“.py”为后缀名，也就是 Python 脚本的后缀名。用作模块的 Python 脚本与其他脚本并没有什么区别。

在 Python 中可以通过导入模块，然后使用模块中提供的函数或者数据。

1. 导入模块

在 Python 中可以使用以下两种方法导入模块或者模块中的函数。

- import 模块名。
- import 模块名 as 新名字。
- from 模块名 import 函数名。

其中，使用 import 是将整个模块导入，而使用 from 则是将模块中某一个函数或者名字导入，而不是整个模块。使用 import 和 from 导入模块还有一个不同：要想使用 import 导入模块中的函数，则必须以模块名＋“.”＋函数名的形式调用函数；而要想使用 from 导入模块中的某个函数，则可以直接使用函数名调用，不必在前面加上模块名称。

以下代码分别演示了使用 import 和 from 导入模块的功能。

```
>>> import math                                  # 使用 import 导入 math 模块
>>>math.sqrt(9)                                 # 使用 math 模块中的 sqrt 函数
3.0'
>>> sqrt(9)                                     # 直接使用 sqrt 名字调用函数
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
NameError: name 'sqrt' is not defined
>>> from math import sqrt                       # 使用 from 导入 math 模块中的 sqrt 函数
>>> sqrt(9)                                     # 直接使用 sqrt 名字调用函数
3.0
```

使用 from 导入模块中的函数后，使用模块中的函数会方便得多，不用在调用函数时使用模块名。如果要想将模块中的所有函数都采用这种方式导入，则可以在 from 中使用 "*" 通配符，表示导入模块中的所有函数。代码如下。

```
>>> from math import sqrt                       # 仅从 math 模块中导入 sqrt
>>> sqrt(9)
3.0
>>> sin(30)                                     # 调用 math 模块中的 sin 函数出错
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
NameError: name sin is not defined
>>> from math import *                          # 重新从 math 模块中导入所有函数
>>> sin(30)                                     # 重新调用 sin 函数
-0.9880316240928618
```

在 Python 2.x 中，还可以通过使用内置函数 reload 重新载入模块。reload 可以在模块被修改的情况下不必关闭 Python 而重新载入模块。在使用 reload 重载模块时，该模块必须已经事先被导入。

在 Python 3 中，reload 函数已经被删除，要重新载入模块，则需要使用 imp 模块中的 reload 函数，具体代码如下。

```
>>> import os                                   # 导入 os 模块
>>> import imp                                  # 导入 imp 模块
>>> imp.reload(os)                              # 重新载入 os 模块
<module 'os' from 'C:\Python32\lib\os.py'>
```

2. 编写一个模块

编写一个 Python 模块是十分简单的事情。

下面是一个简单的模块，该模块中只包含一个函数。这个函数只打印 "I am a module!"。将该模块保存为 mymodule.py。代码如下。

```
def show():                                     # 声明 show 函数
    print('I am a moudle!')
```

编写一个调用函数 show 的脚本，将其保存为 usemodule.py。代码如下。

```
import mymodule                                 # 导入模块
mymodule.show()                                 # 调用模块中的 show 函数
```

脚本运行后输出如下。

```
I am a moudle!
```

除了在模块中声明函数外，还可以在模块中定义变量。模块中的变量同样可以在其他脚本中使用。以下代码在 mymodule.py 模块中添加了一个名为 name 的变量。

```
def show():
    print('I am a moudle!')
name = 'mymodule.py'                    # 在模块中添加一个 name 变量
```

以下代码为修改的 usemodule.py 模块，其中使用了 mymodule.py 中的 name 变量。

```
import mymodule
mymodule.show()
print(mymodule.name)                    # 打印模块中的 name 变量
mymodule.name = 'usemodule.py'          # 将 name 变量重新赋值
print(mymodule.name)
```

脚本运行后输出如下所示。

```
I am a moudle!
mymodule.py
usemodule.py
```

4.5.2 Python 在哪里查找模块

编写好的模块只有被 Python 找到才能被导入。上一节中编写的模块和调用模块的脚本位于同一个目录中，因此不需要进行设置就能被 Python 找到并导入。如果在该目录中新建一个 module 目录，并且把 mymodule.py 转移到 module 目录中，再次在 Windows 的命令窗口中运行 usemodule.py，则输出如下。

```
E:\Python\第 4 章>usemodule.py
Traceback (most recent call last):
  File "usemodule.py", line 1, in <module>
    import mymodule
ImportError: No module named mymodule
```

运行脚本出错，Python 解释器没有找到 mymodule 模块。在导入模块时，Python 解释器首先在当前目录中查找要导入的模块。如果未找到模块，则 Python 解释器会从 sys 模块中 path 变量指定的目录中查找导入模块。如果在以上所有目录中都未找到导入的模块，则会输出出错信息。

以下代码是使用 sys.path 输出 Python 的模块查找路径。

```
>>> import sys
>>> sys.path
['', 'C:\\Windows\\system32\\python32.zip',
'C:\\Python32\\DLLs',
'C:\\Python32\\lib',
'C:\\Python32\\Lib\\site-packages\\pythonwin',
'C:\\Python32',
'C:\\Python32\\lib\\site-packages',
'C:\\Python32\\lib\\site-packages\\win32',
```

```
'C:\\Python32\\lib\\site-packages\\win32\\lib']
```

在脚本中可以向 sys.path 添加模块查找路径。以下所示脚本中，将当前目录下的 module 子目录添加到 sys.path 中，并从 module 目录中导入 mymodule 模块。代码如下。

```
import os
import sys
modulepath = os.getcwd() + '\\module'
sys.path.append(modulepath)
print(sys.path)
import mymodule
mymodule.show()
print(mymodule.name)
mymodule.name = 'usemodule.py'
print(mymodule.name)
```

运行脚本后输出如下。

```
['E:\\Python\\第 4 章', 'C:\\Windows\\system32\\python32.zip',
'C:\\Python32\\DLLs', 'C:\\Python32\\lib',
'C:\\Python32\\Lib\\site-packages\\pythonwin', 'C:\\Python32',
'C:\\Python32\\lib\\site-packages',
'C:\\Python32\\lib\\site-packages\\win32',
'C:\\Python32\\lib\\site-packages\\win32\\lib',
'E:\\Python\\第 4 章\\module', 'C:\\Python32\\module', 'C:\\Python32\\module',
'C:\\Python32\\module',
'C:\\Python32\\module']
I am a moudle!
mymodule.py
usemodule.py
```

从输出可以看出，当前路径也被添加到了 sys.path 路径列表中。这说明 Python 其实是按照 sys.path 中的路径来查找模块的。之所以首先在当前目录中查找，那是因为 Python 解释器在运行脚本前将当前目录添加到 sys.path 路径列表中了。

4.5.3 是否需要编译模块

在上一节的例子中，运行完 usemodule.py 会发现，moudle 目录中除了 mymodule.py 文件外还多了一个 mymodule.pyc 文件。其实，mymodule.pyc 就是 Python 将 mymodule.py 编译成字节码的文件。虽然 Python 是脚本语言，但 Python 可以将脚本编译成字节码的形式。对于模块，Python 总是在第一次调用后就将其编译成字节码的形式，以提高脚本的启动速度。

Python 在导入模块时会查找模块的字节码文件，如果存在则将编译后的模块的修改时间同模块的修改时间相比较。如果两者的修改时间不相符，则 Python 将重新编译模块，以保证两者内容相符。被编译的脚本也是可以直接运行的。

当然，没有必要去刻意编译 Python 脚本。不过，由于 Python 是脚本，如果不想将源文件发布，则可以发布编译后的脚本，这样能起到一定的保护源文件的作用。

对于非模块的脚本，Python 不会在运行脚本后将其编译成字节码的形式。如果想将其编译，可以使用 compile 模块。以下代码可以将 4.5.3 节中的 usemodule.pyc 文件编译成 ".pyc" 文件。

```
# file: compile.py
```

```
#
import  py_compile;                                    # 导入 py_compile 模块
py_compile.compile('usemodule.py','usemodule.pyc');    # 编译 usemodule.py
```

运行 compile.py 后，可以看到当前目录中多了一个 usemodule.pyc 文件。在 Python 3 中，如果在 py_compile.compile 函数中不指定第 2 个参数，则将在当前目录新建一个名为"__pycache__"的目录，并在这个目录中生成"被编译模块名.cpython-32.pyc"的 pytc 字节码文件。

运行 usemodule.pyc 后输出如下。

```
['E:\\Python\\第 4 章', 'C:\\Windows\\system32\\python32.zip',
'C:\\Python32\\DLLs', 'C:\\Python32\\lib',
'C:\\Python32\\Lib\\site-packages\\pythonwin', 'C:\\Python32',
'C:\\Python32\\lib\\site-packages',
'C:\\Python32\\lib\\site-packages\\win32',
'C:\\Python32\\lib\\site-packages\\win32\\lib',
'E:\\Python\\第 4 章\\module', 'C:\\Python32\\module', 'C:\\Python32\\module',
'C:\\Python32\\module',
'C:\\Python32\\module']
I am a moudle!
mymodule.py
usemodule.py
```

可以看到其输出与 4.5.3 节的输出一样。编译后生成的 usemodule.pyc 文件并没有改变程序功能，只是以 Python 字节码的形式存在。

另外，还可以通过 Python 的命令行选项将脚本优化编译。Python 编译的优化选项有以下两个。

- -O　　该选项对脚本的优化不多，编译后的脚本以".pyo"为扩展名。凡是以".pyo"为扩展名的 Python 字节码都是经过优化的。
- -OO　　该选项对脚本优化的程度较大。使用该标志可以使编译的 Python 脚本更小。使用该选项可以导致脚本运行错误，因此，应谨慎使用。

可以通过在命令行中输入以下命令将 usemodule.py 优化编译。

```
python -O compile.py
python -OO compile.py
```

4.5.4 模块也可独立运行

每个 Python 脚本在运行时都有一个__name__属性（name 前后均是两条下画线）。在脚本中通过对__name__属性值的判断，可以让脚本在作为导入模块和独立运行时都可以正确运行。

在 Python 中，如果脚本作为模块被导入，则其__name__属性被设置为模块名；如果脚本独立运行，则其__name__属性被设置为"__main__"。因此可以通过__name__属性来判断脚本的运行状态。

以下所示脚本 mymodule2.py 既可以独立运行，也可以作为模块被其他脚本导入。

```
# file: mymodule2.py
#
def show():
    print 'I am a module!'
```

```
if __name__ == '__main__':
    show()
    print('I am not a module!')
```

以下所示脚本 usemodule2.py 可以调用 mymodule2.py 模块。

```
# file: usemodule2.py
#
import mymodule2
mymodule2.show()
print('my __name__ is', __name__)
```

运行 usemodule2.py 后，输出如下。

```
I am a module!
my __name__ is __main__
```

运行 mymodule2.py 后，输出如下。

```
I am a module!
I am not a module!
```

4.5.5 如何查看模块提供的函数名

如果需要获得导入模块中所有声明的名字、函数等，就可以使用内置的函数 dir()来进行操作。以下所示代码将获得 sys 模块中的名字和函数。

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
'__stderr__', '__stdin__', '__stdout__', '_clear_type_cache', '_current_frames',
'_getframe', '_mercurial', '_xoptions', 'api_version', 'appargv', 'appargvoffset',
'argv', 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
'copyright', 'displayhook', 'dllhandle', 'dont_write_bytecode', 'exc_info',
'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
'float_repr_style', 'getcheckinterval', 'getdefaultencoding',
'getfilesystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount',
'getsizeof', 'getswitchinterval', 'gettrace', 'getwindowsversion', 'hash_info',
'hexversion', 'int_info', 'intern', 'maxsize', 'maxunicode', 'meta_path',
'modules', 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix',
'ps1', 'ps2', 'setcheckinterval', 'setprofile', 'setrecursionlimit',
'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion',
'version', 'version_info', 'warnoptions', 'winver']
```

dir()函数的原型如下。

```
dir( [object])
```

其参数含义如下。

◆ object　　可选参数，要列举的模块名。

如果不向 dir()函数传递参数，那么 dir()函数将返回当前脚本的所有名字列表。如下所示。

```
>>> a = [ 1, 3, 6 ]                    # 定义一个列表
```

```
>>> b = 'python'                          # 定义一个字符串
>>> dir()                                 # 使用 dir()函数获得当前脚本所有名字列表
['__builtins__', '__doc__', '__name__', '__package__', 'a', 'b', 'pywin']
>>> def fun():                            # 定义一个函数
... print( 'Python')
...
>>> dir()                                 # 再次使用 dir()函数
['__builtins__', '__doc__', '__name__', '__package__','a', 'b', 'fun', 'pywin']
```

4.6 用包来管理多个模块

在 Java 中，通过包将不同的类组织在一起。类似的，在 Python 中也提供了包的功能，可以使用包来管理多个模块。使用包的好处在于可以有效避免名字冲突，便于包的维护管理。Python 中的模块包可以通过路径导入模块。

1. 包的组成

包可以看作处于同一目录中的模块。在 Python 中首先使用目录名，然后再使用模块名导入所需要的模块。在包的每个目录中都必须包含一个名为“__init__.py”（init 的前后均是两条下画线）的文件。“__init__.py”可以是一个空文件，仅用于表示该目录是一个包。

“__init__.py”的主要用途是设置“__all__”变量以及所包含的包初始化所需的代码。对于在 from 中使用“*”通配符导入包内所有名字时，在“__init__.py”中设置“__all__”变量可以保证名字的正确导入。

一个简单的 Python 包的目录组成如图 4-1 所示。

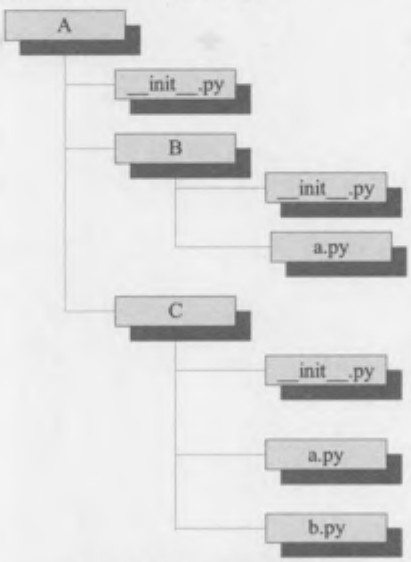


图 4-1 Python 包的组成

在图 4-1 所示的包中，如果需要导入 B 目录中的 a.py 模块，则在 Python 中可以使用以下语句之一。

```
from A.B import a                       # 使用 from 导入模块
```

```
import A.B.a                    # 使用 import 导入模块
```

有了包的概念就可以很好地解决模块查找路径的问题。只要将所有的模块放在当前目录中的某一文件夹内，然后在该文件夹中新建一个空的“__init__.py”文件，就可以通过目录结构的层次导入所需的模块。而不必像前边的例子那样将子目录的路径添加到 sys.path 列表中。

2. 包的内部引用

Python 包中的模块也可能需要相互引用。对于图 4-1 中所示的位于 C 目录中的 b.py，如果要引用同样位于 C 目录中的 a.py，则可以使用以下语句。

```
import  a
```

如果位于 C 目录中的 b.py 要引用位于 B 目录中的 a.py，则需要使用以下语句。

```
from A.B import a
```

4.7 本章小结

本章主介绍了 Python 两个重要的知识点：函数和模块。首先介绍了自定义函数的声明和调用、函数参数的使用、变量的作用域等相关内容。另外，在 Python 中还可以利用 lambda 声明简单的函数，这是一种非常简便的方法。为了让函数方便地重用，本章接着介绍了模块的基本用法，并介绍了通过包来管理多个模块的方法。

在下一章中，将学习用 Python 实现一些常用数据结构和算法的方法。

◆ ◆ ◆

第 5 章　数据结构与算法

本章包括

- 用 Python 操作表
- 用 Python 操作队列
- 用 Python 操作图
- 用 Python 进行排序
- 用 Python 操作栈
- 用 Python 操作树
- 用 Python 进行查找

数据结构是用来描述一种或多种数据元素之间的特定关系，算法是程序设计中对数据操作的描述，数据结构和算法组成了程序。对于简单的任务，只要使用编程语言提供的基本数据类型就足够了；而对于较复杂的任务，就需要使用基本的数据类型来构造出更加复杂的数据结构。

5.1　表、栈和队列

表、堆栈和队列都是基本的线性数据结构。由于 Python 具有设计良好的数据结构，因此其列表可以当作表来使用，而且列表的某些特性与链表相似，因此，在 Python 中，表的实现非常简单。对于栈和队列，则可以自己编写脚本来构建。

5.1.1　表

表是最基本的数据结构，在 Python 中可以使用列表来创建表。而在 C 语言中，一般使用数组来创建表。使用数组所创建的表，在对表中元素进行插入和删除操作时开销较大。当插入一个元素时，要先将该元素后的所有元素，从最后一个元素开始，依次向后移动一个位置。完成元素移动后，再将元素插入到数组中。同样，要删除表中的元素时，首先删除元素，然后将位于该元素之后的元素从前向后，依次向前移动一个位置。

如果一个表含有的元素较多，而要进行插入或删除的位置又比较靠近表的前端，则移动表中元素的操作将耗费大量的时间。为了减少插入和删除元素的线性开销，于是就出现了使用链表代替表。在 C 语言中，链表中不仅保存了数据，还保存了指向下一个元素的指针，如图 5-1 所示。当进行插入操作时，要先将位于插入元素前的元素的指针赋值给插入元素。完成赋值后再将插入元素的地址赋值给位于其前面的元素，如图 5-2 所示。当删除元素时，只需将要删除元素的指针赋值给其前面的元素即可，如图 5-3 所示。



图 5-1　链表

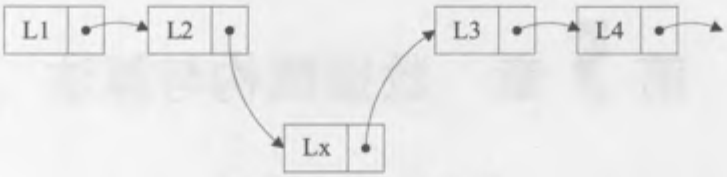


图 5-2　链表的插入操作



图 5-3　链表的删除操作

使用链表，可以降低插入删除、元素的线性开销。然而，由于链表中不仅存储了数据，而且还保存了指向下一个元素的指针，因此使用链表将占用更大的存储空间。

而在 Python 中，列表本身就提供了插入和删除操作。因此，在 Python 中，列表也可以充当链表使用，而不用自己另外编写脚本来构建。

还有一种链表，称之为双向链表，如图 5-4 所示。双向链表中不仅保存了指向下一元素地址的指针，而且还保存了指向其上一个元素地址的指针。相对于单向链表，双向链表需要占用更多的存储空间，但使用双向列表可以完成正序和倒序扫描链表。

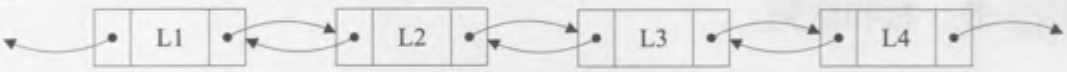


图 5-4　双向链表

5.1.2 栈

栈可以看作在同一位置上进行插入和删除的表，这个位置一般称为栈顶。栈的基本操作是进栈和出栈，栈可以看作一个容器，如图 5-5 所示，先入栈的数据保存在容器底部，后入栈的数据保存在容器顶部。出栈的时候，后入栈的数据先出，而先入栈的数据后出，因此栈有一个特性叫作后进先出（LIFO）。

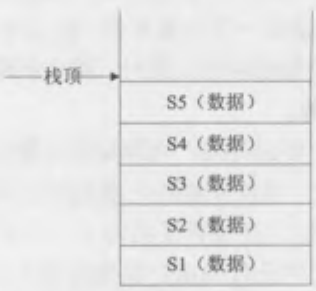


图 5-5　栈

在 Python 中，仍然可以使用列表来存储堆栈数据。通过创建一个堆栈类，实现对堆栈进行操作的方法。例如，进栈 PUSH 方法、出栈 POP 方法，编写检查栈是否为满栈，或者是否为空栈的方法，等等。

下面所示的 pystack.py 在 Python 中创建了一个简单的堆栈结构。

```
# -*- coding:utf-8 -*-
# file: pystack.py
#
```

```
class PyStack:                                          # 堆栈类
    def __init__(self, size = 20):
        self.stack = []                                 # 堆栈列表
        self.size = size                                # 堆栈大小
        self.top = -1                                   # 栈顶位置
    def setSize(self, size):                            # 设置堆栈大小
        self.size = size
    def push(self, element):                            # 元素进栈
        if self.isFull():
            raise StackException('PyStackOverflow')# 如果栈满则引发异常
        else:
            self.stack.append(element)
            self.top = self.top + 1
    def pop(self):                                      # 元素出栈
        if self.isEmpty():
            raise StackException('PyStackUnderflow')    # 如果栈为空则引发异常
        else:
            element = self.stack[-1]
            self.top = self.top - 1
            del self.stack[-1]
            return element
    def Top(self):                                      # 获取栈顶位置
        return self.top
    def empty(self):                                    # 清空栈
        self.stack = []
        self.top = -1
    def isEmpty(self):                                  # 是否为空栈
        if self.top == -1:
            return True
        else:
            return False
    def isFull(self):                                   # 是否为满栈
        if self.top == self.size - 1:
            return True
        else:
            return False

class StackException(Exception):                        #自定义异常类
    def __init__(self,data):
        self.data=data
    def __str__(self):
        return self.data

if __name__ == '__main__':
    stack = PyStack()                                   # 创建栈
    for i in range(10):
        stack.push(i)                                   # 元素进栈
    print(stack.Top())                                  # 输出栈顶位置
    for i in range(10):
        print(stack.pop())                              # 元素出栈
    stack.empty()                                       # 清空栈
    for i in range(21):
        stack.push(i)                                   # 此处将引发异常
```

运行 pystack.py 脚本后，将输出如下所示内容。

```
9
9
8
7
6
5
4
3
2
1
0
Traceback (most recent call last):
  File "C:\Python32\Lib\site-packages\pythonwin\pywin\framework\scriptutils.py",
line 323, in RunScript
    debugger.run(codeObject, __main__.__dict__, start_stepping=0)
  File "C:\Python32\Lib\site-packages\pythonwin\pywin\debugger\__init__.py",
line 60, in run
    _GetCurrentDebugger().run(cmd, globals,locals, start_stepping)
  File "C:\Python32\Lib\site-packages\pythonwin\pywin\debugger\debugger.py",
line 655, in run
    exec(cmd, globals, locals)
  File "E:\Python\第5章\pystack.py", line 4, in <module>
    class PyStack:                                    # 堆栈类
  File "E:\Python\第5章\pystack.py", line 13, in push
    raise StackException('PyStackOverflow')           # 如果栈满则引发异常
StackException: PyStackOverflow
```

5.1.3 队列

队列与栈的结构类似，如图 5-6 所示，不同的是，队列的出队操作是在队首元素进行的删除操作。因此对于队列而言，先入队的元素将先出队。因此队列的特性可以称之为先进先出（FIFO）。



图 5-6 队列

和堆栈类似，在 Python 中同样可以使用列表来构建一个队列，并完成对队列的操作。下面所示的 pyqueue.py 脚本创建了一个简单的队列。

```
# -*- coding:utf-8 -*-
# file: pyqueue.py
#
class PyQueue:                                        # 创建队列
    def __init__(self, size = 20):
        self.queue = []                               # 队列
        self.size = size                              # 队列大小
        self.end = -1                                 # 队尾
    def setSize(self, size):                          # 设置队列大小
        self.size = size
    def In(self, element):                            # 入队
        if self.end < self.size - 1:
            self.queue.append(element)
```

```
            self.end = self.end + 1
        else:
            raise QueueException('PyQueueFull')      # 如果队列满则引发异常
    def Out(self):                                   # 出队
        if self.end != -1:
            element = self.queue[0]
            self.queue = self.queue[1:]
            self.end = self.end - 1
            return element
        else:
            raise QueueException('PyQueueEmpty')     # 如果对列为空则引发异常
    def End(self):                                   # 输出队尾
        return self.end
    def empty(self):                                 # 清除队列
        self.queue = []
        self.end = -1

class QueueException(Exception):                     #自定义异常类
    def __init__(self,data):
        self.data=data
    def __str__(self):
        return self.data

if __name__ == '__main__':
    queue = PyQueue()
    for i in range(10):
        queue.In(i)                                  # 元素入队
    print(queue.End())
    for i in range(10):
        print(queue.Out())                           # 元素出队
    for i in range(20):
        queue.In(i)                                  # 元素入队
    queue.empty()                                    # 清空队列
    for i in range(20):
        print(queue.Out())                           # 此处将引发异常
```

运行 pyqueue.py 脚本后输出如下。

```
9
0
1
2
3
4
5
6
7
8
9
Traceback (most recent call last):
  File "C:\Python32\Lib\site-packages\pythonwin\pywin\framework\scriptutils.py",
line 323, in RunScript
    debugger.run(codeObject, __main__.__dict__, start_stepping=0)
  File "C:\Python32\Lib\site-packages\pythonwin\pywin\debugger\__init__.py",
line 60, in run
    _GetCurrentDebugger().run(cmd, globals,locals, start_stepping)
```

```
  File "C:\Python32\Lib\site-packages\pythonwin\pywin\debugger\debugger.py",
line 655, in run
    exec(cmd, globals, locals)
  File "E:\Python\第5章\pyqueue.py", line 4, in <module>
    class PyQueue:                                    # 创建队列
  File "E:\Python\第5章\pyqueue.py", line 24, in Out
    raise QueueException('PyQueueEmpty')              # 如果对为空则引发异常
QueueException: PyQueueEmpty
```

5.2 树和图

树和前面所讲的表、堆栈和队列等这些线性数据结构不同，树不是线性的。在处理较多数据时，使用线性结构较慢，而使用树结构则可以提高处理速度。不过，相对于线性的表、堆栈和队列等线性数据结构来说，树的构建便显得较为复杂了。

5.2.1 树

树是一种非线性的数据结构，如图 5-7 所示，之所以称之为树，是因为其形状像一颗倒置的树。每棵树都有一个根节点，如图 5-7 所示的树中，Root 为根节点。A、B、C 为 Root 的儿子，Root 为 A、B、C 的父亲。A、B、C 为兄弟。同样，A 为 D、E 的父亲，D、E 为 A 的儿子，D、E 为兄弟。D、E 为 Root 的孙子，Root 为 D、E 的祖父。在树中，如果一个元素没有儿子，则称之为树的叶子。

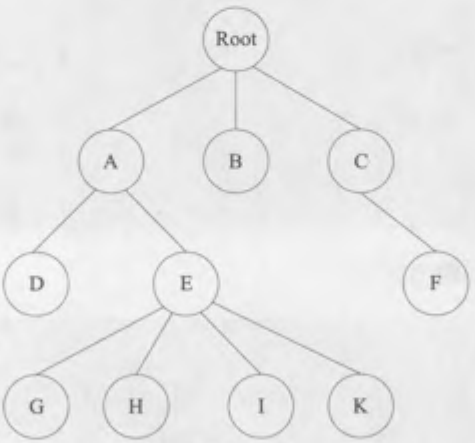


图 5-7 树

在 Python 中，树的实现可以使用列表或者类的方式。使用列表的方式较为简便，但树的构建过程较为复杂。使用类的方式构建树时，需要首先确定树中的节点所能拥有的最大儿子数。因为每个节点所拥有的儿子数量并不一定相同，因此使用类的方法将占用更大的存储空间。

如下所示的 pytree.py 脚本，以列表的形式构建了图 5-7 所示的树。

```
# -*- coding:utf-8 -*-
# file: pytree.py
#
G = [ 'G', [] ]                    # 构造叶子 G，树中每个元素都由该元素的值和该元素的儿子列表组成
```

```
H = [ 'H', [] ]                          # 构造叶子 H
I = [ 'I', [] ]                          # 构造叶子 I
K = [ 'K', [] ]                          # 构造叶子 K
E = [ 'E', [ G, H, I, K] ]               # 构造 E 节点
D = [ 'D', [] ]                          # 构造叶子 D
F = [ 'F', [] ]                          # 构造叶子 F
A = [ 'A', [ D, E] ]                     # 构造 A 节点
B = [ 'B', [] ]                          # 构造叶子 B
C = [ 'C', [ F ] ]                       # 构造 C 节点
Root = [ 'Root', [ A, B, C ] ]           # 构造树根
print(Root)
```

5.2.2 二叉树

二叉树是一类比较特殊的树，在二叉树中每个节点最多只有两个儿子，分为左和右，如图 5-8 所示。相对于树而言，二叉树的构建和使用都要简单得多。

任何一棵树，都可以通过变换转换成一颗二叉树。

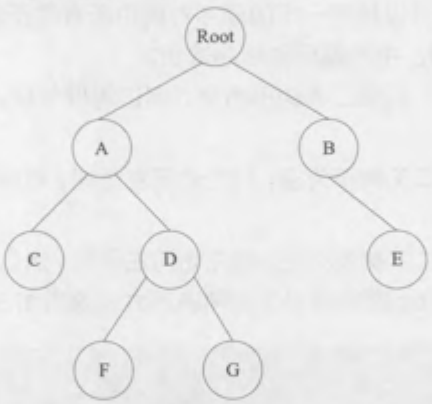


图 5-8 二叉树

在 Python 中，二叉树的构建和树一样，可以使用列表或者类的方式。由于二叉树中的节点具有确定的儿子数，因此，使用类的方式更为简便。下面所示的 pybtree.py 用较为简单的方式生成了如图 5-8 所示的树。

```
# -*- coding:utf-8 -*-
# file: pybtree.py
#
class BTree:                                          # 二叉树节点
    def __init__(self, value):                       # 初始化函数
        self.left = None                              # 左儿子
        self.data = value                             # 节点值
        self.right = None                             # 右儿子
    def insertLeft(self, value):                      # 向左子树插入节点
        self.left = BTree(value)
        return self.left
    def insertRight(self, value):                     # 向右子树插入节点
        self.right = BTree(value)
```

```
        return self.right
    def show(self):                                    # 输出节点数据
        print(self.data)
if __name__ == '__main__':
    Root = BTree('Root')                               # 根节点
    A = Root.insertLeft('A')                           # 向根节点中插入 A 节点
    C = A.insertLeft('C')                              # 向 A 节点中插入 C 节点
    D = A.insertRight('D')                             # 向 A 节点中插入 D 节点
    F = D.insertLeft('F')                              # 向 D 节点中插入 F 节点
    G = D.insertRight('G')                             # 向 D 节点中插入 G 节点
    B = Root.insertRight('B')                          # 向根节点中插入 B 节点
    E = B.insertRight('E')                             # 向 B 节点中插入 E 节点
    Root.show()                                        # 输出节点数据
    Root.left.show()
    Root.right.show()
    A = Root.left
    A.left.show()
    Root.left.right.show()
```

当创建好一棵二叉树后，可以按照一定的顺序对树中所有的元素进行遍历。按照先左后右，树的遍历方法有三种：先序遍历、中序遍历和后序遍历。

其中，先序遍历的次序是：如果二叉树不为空，则访问根节点，然后访问左子树，最后访问右子树；否则，程序退出。

中序遍历的次序是：如果二叉树不为空，则先访问左子树，然后访问根节点，最后访问右子树；否则，程序退出。

后序遍历的次序是：如果二叉树不为空，则先访问左子树，然后访问右子树，最后访问根节点。

下面所示的 TreeTraversal.py 脚本使用了三种遍历方式遍历图 5-8 所示的树。

```
# -*- coding:utf-8 -*-
# file: TreeTraversal.py
#
class BTree:                                           # 二叉树节点
    def __init__(self, value):                        # 初始化函数
        self.left = None                               # 左儿子
        self.data = value                              # 节点值
        self.right = None                              # 右儿子
    def insertLeft(self, value):                       # 向左子树插入节点
        self.left = BTree(value)
        return self.left
    def insertRight(self, value):                      # 向右子树插入节点
        self.right = BTree(value)
        return self.right
    def show(self):                                    # 输出节点数据
        print(self.data)
def preorder(node):                                    # 先序遍历
    if node.data:
        node.show()
        if node.left:
            preorder(node.left)
        if node.right:
```

```
            preorder(node.right)
def inorder(node):                                    # 中序遍历
    if node.data:
        if node.left:
            inorder(node.left)
        node.show()
        if node.right:
            inorder(node.right)
def postorder(node):                                  # 后序遍历
    if node.data:
        if node.left:
            postorder(node.left)
        if node.right:
            postorder(node.right)
        node.show()
if __name__ == '__main__':
    Root = BTree('Root')                              # 构建树
    A = Root.insertLeft('A')
    C = A.insertLeft('C')
    D = A.insertRight('D')
    F = D.insertLeft('F')
    G = D.insertRight('G')
    B = Root.insertRight('B')
    E = B.insertRight('E')
    print('**************************')
    print('Binary Tree Pre-Traversal')
    print('**************************')
    preorder(Root)                                    # 对树进行先序遍历
    print ('**************************')
    print('Binary Tree In-Traversal')
    print('**************************')
    inorder(Root)                                     # 对树进行中序遍历
    print('**************************')
    print('Binary Tree Post-Traversal')
    print('**************************')
    postorder(Root)                                   # 对树进行后序遍历
```

运行 TreeTraversal.py 脚本后输出如下。

```
**************************
Binary Tree Pre-Traversal
**************************
Root
A
C
D
F
G
B
E
**************************
Binary Tree In-Traversal
**************************
C
A
F
D
```

```
G
Root
B
E
*************************
Binary Tree Post-Traversal
*************************
C
F
G
D
A
E
B
Root
```

5.2.3 图

图是非线性的数据结构，图是由顶点和边组成的。如果图中的顶点是有序的，那么图是有方向的，称之为有向图，如图 5-9 所示；否则，图是无方向的，称之为无向图。在图中，由顶点组成的序列称之为路径。

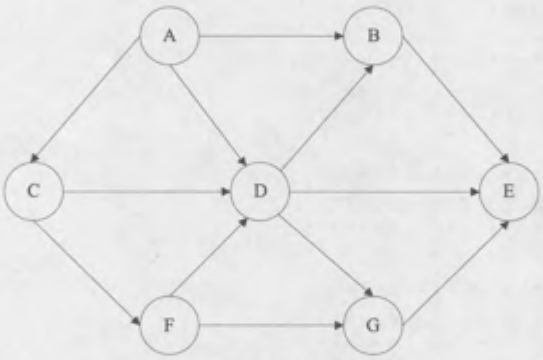


图 5-9　有向图

图和树相比，少了树那样明显的层次结构。

在 Python 中，可以采用字典的方式来创建图，图中的每个元素都是字典中的键，该元素所指向的图中其他元素组成键的值。

与树一样，对于图来说，也可以对其进行遍历。除了遍历以外，还可以在图中搜索所有的从一个顶点到另一个顶点的路径。

图中的每一顶点可以看作一个城市，路径可以看作城市到城市之间的公路。因此，通过搜索所有的路径，可以找到一个顶点到另一个顶点的最短路径，即城市到城市间的最短路线。

下面所示的 pygraph.py 是使用字典的方式构建的如图 5-9 所示的有向图，并搜索图中的路径。

```
# -*- coding:utf-8 -*-
# file: pygraph.py
#
def searchGraph(graph, start, end):                    # 搜索树
    results = []
    generatePath(graph, [start], end, results)         # 生成路径
    results.sort(key=lambda x:len(x))                  # 按路径长短排序
```

```
    return results
def generatePath(graph, path, end, results):                    # 生成路径
    state = path[-1]
    if state == end:
        results.append(path)
    else:
        for arc in graph[state]:
            if arc not in path:
                generatePath(graph, path + [arc], end, results)
if __name__ == '__main__':
    Graph = {'A': ['B', 'C', 'D'],                              # 构建树
             'B': ['E'],
             'C': ['D', 'F'],
             'D': ['B', 'E', 'G'],
             'E': [],
             'F': ['D', 'G'],
             'G': ['E']}
    r = searchGraph(Graph, 'A','D')                             # 搜索A到D的所有路径
    print('************************')
    print('    path A to D')
    print('************************')
    for i in r:
        print(i)
    r = searchGraph(Graph, 'A','E')                             # 搜索A到E的所有路径
    print('************************')
    print('    path A to E')
    print('************************')
    for i in r:
        print(i)
    r = searchGraph(Graph, 'C','E')                             # 搜索C到E的所有路径
    print('************************')
    print('    path C to E')
    print('************************')
    for i in r:
        print(i)
```

运行 pygraph.py 脚本后，将输出如下内容。

```
************************
    path A to D
************************
['A', 'D']
['A', 'C', 'D']
['A', 'C', 'F', 'D']
************************
    path A to E
************************
['A', 'B', 'E']
['A', 'D', 'E']
['A', 'C', 'D', 'E']
['A', 'D', 'B', 'E']
['A', 'D', 'G', 'E']
['A', 'C', 'D', 'B', 'E']
['A', 'C', 'D', 'G', 'E']
['A', 'C', 'F', 'D', 'E']
['A', 'C', 'F', 'G', 'E']
['A', 'C', 'F', 'D', 'B', 'E']
['A', 'C', 'F', 'D', 'G', 'E']
```

```
************************
    path C to E
************************
['C', 'D', 'E']
['C', 'D', 'B', 'E']
['C', 'D', 'G', 'E']
['C', 'F', 'D', 'E']
['C', 'F', 'G', 'E']
['C', 'F', 'D', 'B', 'E']
['C', 'F', 'D', 'G', 'E']
```

5.3 查找与排序

查找和排序是最基本的算法，在很多脚本中都会用到查找和排序。其实，在前面各章的例子中，已经多次用到 Python 提供的查找和排序函数查找字符串中的子字符串。尽管 Python 提供的用于查找和排序的函数能够满足绝大多数需求，但还是有必要了解最基本的查找和排序算法，以便在有特殊需求的情况下，可以自己编写查找、排序脚本。

5.3.1 查找

基本的查找方法有顺序查找、二分查找和分块查找等。其中，顺序查找是最简单的查找方法，就是按数据排列的顺序依次查找，直到找到所查找的数据为止（可查看数据表都未找到数据）。

二分查找是首先对要进行查找的数据进行排序，有按大小顺序排好的 9 个数字，如图 5-10 所示。如果要查找数字 5，则首先与中间值 10 进行比较，5 小于 10，于是对序列的前半部分 1～9 进行查找。此时，中间值为 5，恰好为要找的数字，查找结束。



图 5-10　二分查找

分块查找，是介于顺序查找和二分查找之间的一种查找方法。使用分块查找时，首先对查找表建立一个索引表，然后再进行分块查找。建立索引表时，首先对查找表进行分块，要求“分块有序”，即块内关键字不一定有序，但分块之间有大小顺序。索引表是抽取各块中的最大关键字及其起始位置构成的，如图 5-11 所示。

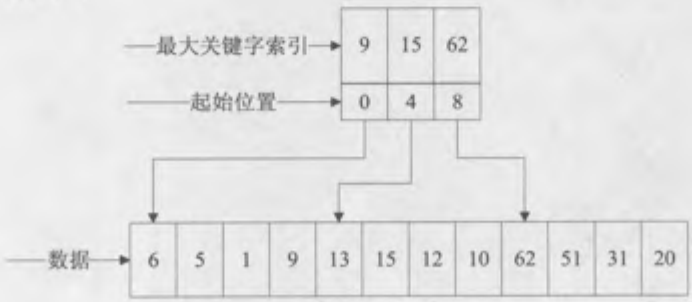


图 5-11　分块查找

分块查找分两步进行，首先查找索引表，因为索引表是有序的，查找索引表时可以使用二分查找法进行。查找完索引表以后，就确定了要查找的数据所在的分块，然后在该分块中再进行顺序查找。

下面所示的 pyBinarySearch.py 脚本是对一个有序列表使用二分查找。

```
# -*- coding:utf-8 -*-
# file: pyBinarySearch.py
#
def BinarySearch(l, key):                              # 二分查找
    low = 0
    high = len(l) - 1
    i = 0
    while (low <= high):
        i = i + 1
        mid = (high + low) // 2
        if (l[mid] < key):
            low = mid + 1
        elif (l[mid] > key):
            high = mid - 1
        else:
            print('use %d time(s)' % i)
            return mid
    return -1
if __name__ == '__main__':
    l = [1, 5, 6, 9, 10, 51, 62, 65, 70]               # 构造列表
    print(BinarySearch(l, 5))                          # 在列表中查找
    print(BinarySearch(l, 10))
    print(BinarySearch(l, 65))
    print(BinarySearch(l, 70))
```

运行脚本后输出如下。

```
use 2 time(s)
1
use 1 time(s)
4
use 3 time(s)
7
use 4 time(s)
8
```

5.3.2 排序

相对于查找来说，排序要复杂得多，排序的方法也较多，常用的排序方法有冒泡法排序、希尔排序、二叉树排序和快速排序等。其中，二叉树排序是比较有意思的一种排序方法，而且也便于操作。二叉树排序的过程主要是二叉树的建立和遍历的过程。例如，有一组数据“3, 5 , 7, 20, 43, 2, 15, 30”，则二叉树的建立过程如下。

step 1 首先将第一个数据 3 放入根节点。

step 2 将数据 5 与根节点中的数据 3 比较，由于 5 大于 3，因此应将 5 放入 3 的右子树中。

step 3 将数据 7 与根节点中的数据 3 比较，由于 7 大于 3，因此应将 7 放入 3 的右子树中，由于 3 已经有右儿子 5，所以将 7 与 5 进行比较，因为 7 大于 5，所以应将 7 放入 5 的右子

树中。

step 4 将数据 20 与根节点 3 进行比较，由于 20 大于 3，因此应将 20 放入 3 的右子树，重复比较，最终将 20 放到 7 的右子树中。

step 5 将数据 43 与树中的节点值进行比较，最终将其放入 20 的右子树中。

step 6 将数据 2 与根节点 3 进行比较，由于 2 小于 3，因此应将 2 放入 3 的左子树。

step 7 同样的对数据 15 和 30 进行处理，最终形成如图 5-12 所示的树。

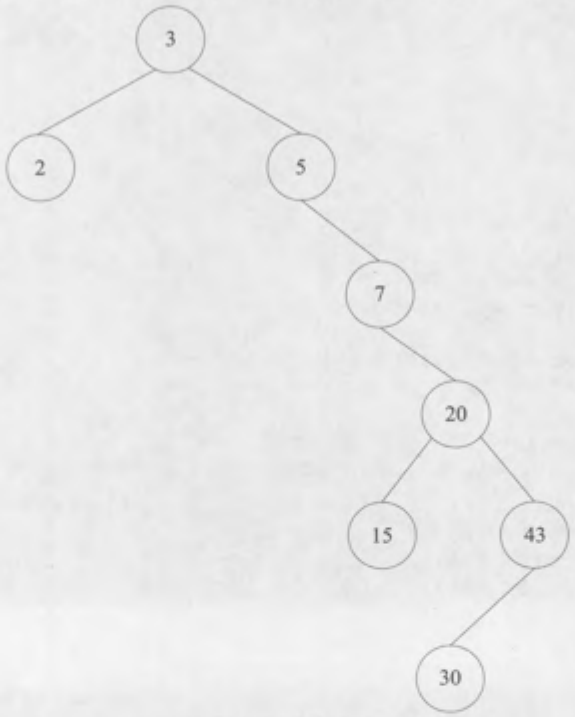


图 5-12 二叉树排序

当树创建好后，对树进行中序遍历，得到的遍历结果就是对数据从小到大排序。如果要从大到小进行排序，则可以先从右子树开始进行中序遍历。

下面所示的 pySort.py 脚本是采用二叉树排序的方式对数据进行排序。

```
# -*- coding:utf-8 -*-
# file: pySort.py
#
class BTree:                                    # 二叉树节点
    def __init__(self, value):                 # 初始化函数
        self.left = None                        # 左儿子
        self.data = value                       # 节点值
        self.right = None                       # 右儿子
    def insertLeft(self, value):                # 向左子树插入节点
        self.left = BTree(value)
        return self.left
    def insertRight(self, value):               # 向右子树插入节点
        self.right = BTree(value)
        return self.right
```

```
    def show(self):                                    # 输出节点数据
        print(self.data)
def inorder(node):                                     # 中序遍历
    if node.data:
        if node.left:
            inorder(node.left)
        node.show()
        if node.right:
            inorder(node.right)
def rinorder(node):                                    # 中序遍历,先遍历右子树
    if node.data:
        if node.right:
            rinorder(node.right)
        node.show()
        if node.left:
            rinorder(node.left)
def insert(node, value):
    if value > node.data:
        if node.right:
            insert(node.right, value)
        else:
            node.insertRight(value)
    else:
        if node.left:
            insert(node.left, value)
        else:
            node.insertLeft(value)
if __name__ == '__main__':
    l = [3, 5 , 7, 20, 43, 2, 15, 30]
    Root = BTree(l[0])                                 # 根节点
    node = Root
    for i in range(1, len(l)):
        insert(Root, l[i])
    print('******************************')
    print('       从小到大')
    print('******************************')
    inorder(Root)
    print('******************************')
    print('       从大到小')
    print('******************************')
    rinorder(Root)
```

运行 pySort.py 脚本后，输出如下所示的排序结果。

```
******************************
       从小到大
******************************
2
3
5
7
15
20
30
43
******************************
       从大到小
```

```
******************************
43
30
20
15
7
5
3
2
```

5.4 本章小结

本章主要介绍了 Python 在处理基本数据结构方法时的脚本编写方法。由于 Python 拥有设计良好的数据结构，因此，通过列表就可以方便地完成表、栈、队列、树、图等数据结构的操作。只要掌握了 Python 列表数据类型的使用，编写操作数据结构中的表、栈、队列、树、图等脚本就比较简单了。本章最后还介绍了用 Python 编写查找与排序的脚本的方法。

在下一章中，将重点介绍 Python 的面向对象特性。

◆ ◆ ◆

第 6 章　面向对象的 Python

本章包括

- Python 中的面向对象思想
- 在 Python 中定义和使用类
- 类的继承
- 在模块中定义类
- 认识类和对象
- 类的属性和方法
- 在类中重载方法和运算符

Python 是一种面向对象的编程语言，不过，Python 与 C++一样，还支持面向过程的程序设计。在 Python 中完全可以使用函数、模块等方式来完成工作。但是，当使用 Python 编写一个较为庞大的项目时，则应该考虑使用面向对象的方法，以便更好地对项目进行管理。

6.1　面向对象编程概述

面向对象程序设计（Object Oriented Programming）简称 OOP，是与面向过程的程序设计不同的另一种编程架构。本书前面各章的例子基本都是面向过程，而非面向对象的。

6.1.1　Python 中的面向对象思想

面向对象程序设计是从 20 世纪 90 年代开始流行的一种编程方法，强调对象的“抽象”、“封装”、“继承”和“多态”。面向对象程序设计方法的基本思想是将任何事物都当作对象，是其所属于对象类的一个实例。对于复杂的对象则将其划分成简单的对象，由这些简单的对象以某种方式组合而形成复杂的对象。每一个对象都有其相对应的对象类，属于同一对象类的对象具有相同的属性及方法。

对象以对象类的形式将其内部的数据或者方法进行封装。对象与对象之间只是相互传递数据，而不能访问其他对象的内部，对象的内部对其他对象而言是不可见的。不同的对象类之间可以通过继承的形式来拥有其他对象的属性和方法，从而形成父子关系。

面向对象程序设计方法的基本过程如下。

step 1　确定对象及其属性和方法。
step 2　分析对象之间的联系，确定其通信机制。
step 3　将具有共同特征的对象抽象为对象类。
step 4　设计、实现类，并确定类相互间的继承关系。
step 5　创建对象实例，实现对象间的相互联系。

例如，可以将人作为一个对象类。每一个具体的人，如张三，则是一个对象实例。每个人都具有姓名、性别、年龄和身高等特征，可以将这些特征抽象为对象类的属性。

Python 完全采用了面向对象程序设计的思想。在 Python 中，可以使用类建立一个对象模型，以及对象所拥有的属性和方法。该模型能够较好地反映事物的本质，以及其相互之间的关系，其本质是更接近于人类认知事物所采用的计算模型。

Python 是真正面向对象的脚本语言，虽然其与 C++的类机制有所区别，但 Python 能够保证对类的最重要功能的支持，如类的继承、基类的重载等。

在 Python 中，对象概念比较广泛，对象不一定就是类的实例。Python 的内置数据类型如字符串、列表、字典等，它们都不是类，但却具有一些和类相似的语法。例如，使用“.”操作符来使用内置类型的某些方法。

6.1.2 类和对象

类是面向对象程序设计的基础。类具有抽象性、封装性、继承性和多态性。

- 类的抽象性：类是对具有共同方法和属性的一类对象的描述。
- 类的封装性：类将属性和方法封装，外部是不可见的，只有通过类提供的接口才能与属于类的实例对象进行信息交换。
- 类的继承性：类可以由已有的类派生。派生出来的类拥有父类的方法和属性。
- 类的多态性：类可以根据不同的参数类型调用不同的方法。同一个方法可以处理不同类型的参数。实际上，Python 的内部已经很好地实现了多态，在 Python 中，使用类不需要考虑太多不同类型之间数据的处理问题。

每个类都有自己的属性和方法。类的属性，实际上就是类内部的变量；而类的方法，则是在类内部定义的函数。

对象是具体的事物，是类的实例化结果。每个对象的属性值可能不同，但所有由同一类实例化得来的对象都拥有共同的属性和方法。在程序中，由类实例化生成对象，然后使用对象的方法进行操作，从而来完成任务。一个类可以实例化生成多个对象。类与对象的关系如图 6-1 所示。

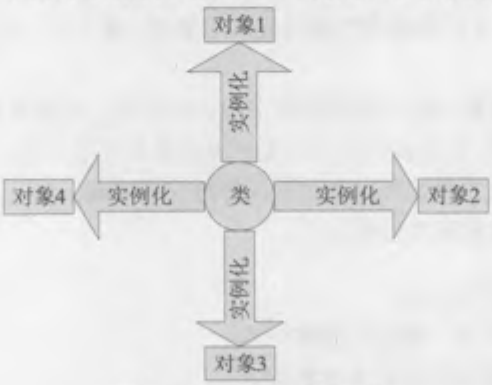


图 6-1 类与对象的关系

6.2 在 Python 中定义和使用类

由于 Python 对面向对象程序设计有着良好的支持，因此在 Python 中定义和使用类并不复杂。

类的定义和使用跟函数的定义和使用有很多相似之处。

6.2.1 类的定义

在 Python 中，类的定义与函数的定义类似，所不同的是，类的定义是使用关键字“class”。与函数定义相同的是，在定义类时也要使用缩进的形式，以表示缩进的语句属于该类。类的定义形式如下。

```
class <类名>:
    <语句 1>
    <语句 2>
    …
    <语句 3>
```

与函数定义相同，在使用类之前必须先定义类。类的定义一般放在脚本的头部。在 Python 中也可以在 if 语句的分支或者函数定义中定义类。下面的代码定义了一个 human 类，并定义了相关属性。

```
>>> class human:                    # 定义 human 类
...  age = 0                        # 定义 age 属性
...  sex = ''                       # 定义 sex 属性
...  height = 0                     # 定义 height 属性
...  weight = 0                     # 定义 weight 属性
...  name = ''                      # 定义 name 属性
```

类还可以通过继承的形式来进行定义。通过类继承来定义类的基本形式如下。

```
class <类名>(父类名):
    <语句 1>
    <语句 2>
    …
    <语句 3>
```

其中圆括号中的父类名就是要继承的类。关于继承将在后面的章节中讲解，此处给出一个简单的例子。以下所示的代码是通过继承 human 类来生成一个新类。

```
>>> class student(human):           # 通过继承 human 类创建 student 类
...  school = ''                    # 定义新属性 school
...  number = 0                     # 定义新属性 number
...  grade = 0                      # 定义新属性 grade
...
```

上述通过 human 继承而来的 student 类，自动继承了 human 中的属性。同时，在上面的代码中又为 student 类定义了其他的属性。

类定义后就产生了一个名字空间，与函数类似。在类内部使用的属性，相当于函数中的变量名，还可以在类的外部继续使用。类的内部与函数的内部一样，相当于一个局部作用域。不同类的内部也可以使用相同的属性名。

6.2.2 类的使用

类在定义后必须先实例化才能使用。类的实例化与函数调用类似，只要使用类名加圆括号的形式就可以实例化一个类。

类实例化后会生成一个对象。一个类可以实例化成多个对象，对象与对象之间并不会相互影响。类实例化后就可以使用其属性和方法等。下面的代码首先定义了一个 book 类，然后将其实例化。

```
>>> class book:                          # 定义 book 类
... author = ''                          # 定义 author 属性
... name = ''                            # 定义 name 属性
... pages = 0                            # 定义 pages 属性
... price = 0                            # 定义 press 属性
... press = ''
...
>>> a = book()                           # book 类实例化
>>> a                                    # 查看对象 a
<class __main__.book at 0x011205A0>
>>> a.author                             # 访问 author 属性
''
>>> a.pages                              # 访问 pages 属性
0
>>> a.price                              # 访问 price 属性
0
>>> a.author = 'Tom'                     # 设置 author 属性
>>> a.pages = 300                        # 设置 pages 属性
>>> a.price = 25                         # 设置 price 属性
>>> a.author                             # 重新访问 author 属性
'Tom'
>>> a.pages                              # 重新访问 pages 属性
300
>>> a.price                              # 重新访问 price 属性
25
>>> b = book()                           # 将 book 实例化生成 b 对象
>>> b.author                             # 访问 b 对象的 author 属性
''
>>> b.price                              # 访问 b 对象的 price 属性
0
>>> b.author = 'Jack'                    # 设置 b 对象的 author 属性
>>> b.author
'Jack'
>>> b.price = 15                         # 设置 b 对象的 price 属性
>>> b.price
15
>>> a.price                              # a 对象的 price 属性并没有改变
25
>>> a.author                             # a 对象的 author 属性也没有改变
'Tom'
>>> b.pages                              # 访问 b 对象的 pages 属性
0
>>> a.pages                              # 访问 a 对象的 pages 属性
```

300

上述例子只定义了类的属性，并在类实例化后重新设置其属性。从代码中可以看出，类的实例化相当于调用一个函数，这个函数就是类。函数返回一个类的实例对象，返回后的对象就具有了类所定义的属性。上述例子生成了两个 book 实例对象，设置其中一个对象的属性，并不会影响到另一个对象的属性，也就是说，这两个对象是相互独立的。

在 Python 中需要注意的是，虽然类首先需要实例化，然后才能使用其属性。但实际上当创建一个类以后就可以通过类名访问其属性。如果直接使用类名修改其属性，那么将影响已经通过该类实例化的其他对象。演示代码如下。

```
>>> class A:                          # 定义类 A
... name = 'A'                        # 定义属性 name 将其赋值为 'A'
... num = 2                           # 定义 num 属性将其赋值为 2
...
>>> A.name                            # 直接使用类名访问类的属性
'A'
>>> A.num                             # 直接使用类名访问类的属性
2
>>> a = A()                           # 生成 a 对象
>>> a.name                            # 查看 a 的 name 属性
'A'
>>> b = A()                           # 生成 b 对象
>>> b.name                            # 查看 b 的 name 属性
'A'
>>> A.name = 'B'                      # 使用类名修改 name 属性
>>> a.name                            # a 对象的 name 属性被修改
'B'
>>> b.name                            # b 对象的 name 属性也被修改
'B'
```

6.3 类的属性和方法

每一个类都具有自己的属性和方法。属性和方法是面向对象程序设计所独有的概念。属性是类所封装的数据，而方法则是类对数据进行的操作。

6.3.1 类的属性

在 6.2 节中已经简单地定义和使用了类的属性，类的属性实际上是类内部的变量。6.2 节的例子中使用了类的属性，确切地说，是类的公有属性。

在 6.2 节的例子中，类的外部可以设置其属性的值，在某些情况下，可能不希望在类的外部对其属性进行操作。此时就可以使用类的私有属性。

数据保护是面向对象程序设计所特有的，在面向过程的程序设计中并没有数据保护的概念。类中的私有属性是不能在类的外部进行操作的，这便起到了对属性的保护作用。Python 与 C++不同，在类的内部声明一个私有成员时不需要使用 private 关键字，在 Python 中，是通过类中属性的命名形式来表示类属性是公有还是私有的。

在 Python 中，如果类中的属性是以双下画线开始的，则该属性为类的私有属性，不能在类的外部被使用或者访问。下面是一个私有属性的命名形式。

```
__priavte_attrs                          # 以双下画线开始
```

如果在类内部的方法中使用类的私有属性，则应该用下面的方式调用。

```
self.__priavte_attrs                     # 应该在私有属性名前加上 "self."
```

下面的代码是 6.2 节中的 book 类进行修改，将其部分属性改为私有属性。

```
>>> class book:                          # 定义一个类
...     __author = ''                    # 类的私有属性
...     __name = ''                      # 类的私有属性
...     __page = 0                       # 类的私有属性
...     price = 0                        # 类的公有属性
...     __press = ''                     # 类的私有属性
...
>>> a = book()                           # 实例化 book 类生成 a 对象
>>> a.__author                           # 试图访问对象的__author 私有属性，结果导致错误
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AttributeError: 'book' object has no attribute '__author'
>>> a.price                              # 访问对象的 price 公有属性
0
>>> a.price = 20                         # 修改 price 属性
>>> a.price                              # price 属性改变了
20
>>> a.__name                             # 试图访问对象的__name 私有属性，结果导致错误
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AttributeError: 'book' object has no attribute '__name'
>>> a.__page
Traceback (most recent call last):       # 试图访问对象的__page 私有属性，结果导致错误
  File "<interactive input>", line 1, in <module>
AttributeError: 'book' object has no attribute '__page'
```

可以看到，在类定义的时候，凡是以双下画线开始的属性不能在类的外部访问，当然也不能修改。如果要修改类的私有属性值或者获取其值，则可以通过使用类提供的方法来完成。

6.3.2 类的方法

类的方法实际上就是类内部使用 def 关键字定义的函数。定义类的方法与定义一个函数基本相同，在类的方法中同样也要使用缩进。

1. 定义类的方法

在类的内部使用 def 关键字可以为类定义一个方法。与函数定义不同的是，类的方法必须包含参数“self”，且“self”必须为第一个参数。下面的代码是为 6.3.1 节中的 book 类添加 show 方法和 setname 方法。

```
>>> class book:                          # 定义 book 类
...     __author = ''                    # 类的私有属性
...     __name = ''                      # 类的私有属性
...     __page = 0                       # 类的私有属性
...     price = 0                        # 类的公有属性
...     __press = ''                     # 类的私有属性
...     def show(self):                  # 定义类的 show 方法，其参数必须为 self
...         print(self.__author)         # 输出类的私有属性
...         print(self.__name)           # 输出类的私有属性
...     def setname(self,name):          # 定义类的 setname 方法，其有两个参数
...         self.__name = name           # 设置类的__name 私有属性
...
>>> a = book()                           # 生成类的实例
>>> a.show()                             # 调用 show 方法，输出为空，因为其私有属性都为空


>>>
>>> a.setname('Tom')                     # 调用 setname 方法，但只向其传递一个参数
>>> a.show()                             # 调用 show 方法

Tom
```

与类的属性相同，类的方法也可以是类私有的，类的私有方法不能在类的外部调用。和类的私有属性命名相同，类的私有方法名也要以双下画线开始。

类的私有方法只能在类的内部调用，而不能在类的外部调用。另外，在类的内部调用其私有方法时，要使用“self.私有方法名”的形式。下面的代码是为 book 类添加一个名为 check 的私有方法，并且修改 show()方法。

```
>>> class book:                          # 定义 book 类
...     __author = ''                    # 类的私有属性
...     __name = ''                      # 类的私有属性
...     __page = 0                       # 类的私有属性
...     price = 0                        # 类的公有属性
...     __press = ''                     # 类的私有属性
...     def __check(self,item):          # 定义__check 私有方法
...         if item == '':               # 判断 item 是否为空
...             return 0                 # 为空则返回 0
...         else:
...             return 1                 # 否则返回 1
...     def show(self):                  # 修改 show 方法
...         if self.__check(self.__author):# 判断__author 私有属性是否为空
...             print(self.__author)     # 不为空则输出其值
...         else:
...             print('No value')        # 否则输出 "No value"
...         if self.__check(self.__name):  # 判断私有属性__check 是否为空
...             print(self.__name)       # 不为空则输出其值
...         else:
...             print('No value')        # 否则输出 "No value"
```

```
... def setname(self,name):                    # 定义 setname 方法
...     self.__name = name
...
>>> a = book()                                 # 类的实例化
>>> a.show()                                   # 调用 show 方法
No value
No value
>>> a.setname('Tom')                           # 调用 setname 方法
>>> a.show()                                   # 重新调用 show 方法，看到输出值已经改变
No value
Tom
>>> a.__check()                                # 调用类的私有方法，结果出错
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AttributeError: 'book' object has no attribute '__check'
```

2. 类的专有方法（Special Methods）

在 Python 中，类中有一些以双下画线开始并且以双下画线结束的方法，我们称之为类的专有方法。专有方法是针对类的特殊操作的一些方法，例如，在类实例化时将调用__init__方法。部分类的专有方法如表 6-1 所示。

表 6-1　类的专有方法

方法名	描述
__init__	构造函数，生成对象时调用
__del__	析构函数，释放对象时调用
__add__	加运算
__mul__	乘运算
__cmp__	比较运算
__repr__	打印，转换
__setitem__	按照索引赋值
__getitem__	按照索引获取值
__len__	获得长度
__call__	函数调用

以下代码修改 book 类，使用__init__方法为对象的属性赋初始值。

```
>>> class book:                                # 定义 book 类
...  __author = ''                             # 类的私有属性
...  __name = ''                               # 类的私有属性
...  __page = 0                                # 类的私有属性
...  price = 0                                 # 类的公有属性
...  __press = ''                              # 类的私有属性
...  def __check(self,item):                  # 定义__check 私有方法
...      if item == '':                        # 判断 item 是否为空
...          return 0                          # 为空则返回 0
...      else:
...          return 1                          # 否则返回 1
```

```
... def show(self):                          # 修改 show 方法
...     if self.__check(self.__author):# 判断__author 私有属性是否为空
...         print(self.__author)             # 不为空则输出其值
...     else:
...         print('No value')               # 否则输出 "No value"
...     if self.__check(self.__name):   # 判断私有属性__check 是否为空
...         print(self.__name)               # 不为空则输出其值
...     else:
...         print('No value')               # 否则输出 "No value"
... def setname(self,name):                  # 定义 setname 方法
...     self.__name = name
... def __init__(self,author,name):      # 使用__init__方法，为__author 及__name 赋初值
...     self.__author = author
...     self.__name = name
...
>>> a = book('Tom','A Wonderfull Book')# 实例化，为__author 及__name 赋初值
>>> a.show()                                 # 调用 show 方法
Tom
A Wonderfull Book
>>> a.setname('About Jack')                  # 重新设置__name 私有属性
>>> a.show()                                 # 调用 show 方法
Tom
About Jack
```

6.4 类的继承

一个新类可以通过继承来获得已有类的方法及属性等。通过继承而来的类也可以自己定义新的方法和属性。

6.4.1 使用继承

在本章前面介绍类的定义时已经提到如何通过继承来获得一个新类。新类可以继承父类的公有属性和公有方法，但是不能继承父类的私有属性和私有方法。下面所示的代码是通过继承 book 类来创建一个名为 student 的类。

```
>>> class book:                              # 定义 book 类
...     __author = ''                        # 类的私有属性
...     __name = ''                          # 类的私有属性
...     __page = 0                           # 类的私有属性
...     price = 0                            # 类的公有属性
...     __press = ''                         # 类的私有属性
...     def __check(self,item):              # 定义__check 私有方法
...         if item == '':                   # 判断 item 是否为空
...             return 0                     # 为空则返回 0
...         else:
...             return 1                     # 否则返回 1
...     def show(self):                      # 修改 show 方法
```

```
...     if self.__check(self.__author):     # 判断__author 私有属性是否为空
...         print(self.__author)            # 不为空则输出其值
...     else:
...         print('No value')               # 否则输出 "No value"
...     if self.__check(self.__name):       # 判断私有属性__check 是否为空
...         print(self.__name)              # 不为空则输出其值
...     else:
...         print('No value')               # 否则输出 "No value"
... def setname(self,name):                 # 定义 setname 方法
...     self.__name = name
... def __init__(self,author,name):  # 使用__init__方法，为__author 及__name 赋初值
...     self.__author = author
...     self.__name = name
...
>>> class student(book):                    # 通过继承创建 student 类
... __class = ''
... __grade = ''
... __sname = ''
... def showinfo(self):                     # 定义一个新的 showinfo 方法
...     self.show()                         # 此处仅调用 book 类的 show 方法
...
>>> b = student('Jack','Big Book')          # student 类继承了 book 类的__init__方法
>>> b.showinfo()                            # 调用 student 类的 showinfo 方法
Jack
Big Book
>>> b.show()                                # 调用 book 类的 show 方法
Jack
Big Book
```

如果在定义类的时候试图使用父类的私有属性或者私有方法将会导致错误，代码如下。

```
>>> class student(book):
... def showall(self):
        # 此处调用了父类的__check 方法，且使用了父类的__name 私有属性
...     if self.__check(self.__name):
...         print(self.__name)
...     else:
...         print('No Value')
...
>>> c = student('Tom','Big Book')
>>> c.showall()                             # 调用 showall 方法时出错
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "<interactive input>", line 3, in showall
AttributeError: 'book' object has no attribute '_student__check'
```

6.4.2 Python 的多重继承

多重继承是指创建的类同时拥有几个类的属性和方法。多重继承与单重继承不同的是，在类名后边的圆括号中可以包含多个父类名，父类名之间以逗号隔开。通过多重继承创建一个新类的一般形式如下。

```
>>> class 新类名(父类 1,父类 2,……,父类 n):
```

```
        <语句 1>
        <语句 2>
        …
        <语句 3>
```

使用多重继承需要注意圆括号中父类名字的顺序。如果父类中有相同的方法名，而在类中使用时未指定父类名，则 Python 解释器将从左至右搜索。

以下代码首先定义了两个类，然后通过多重继承创建一个新类。

```
>>> class A:                                    # 定义类 A
... name = 'A'                                  # 定义其 name 属性
... __num = 1                                   # 定义其__num 属性
... def show(self):                             # 定义 show 方法
...     print( self.name)
...     print(self.__num)
... def setnum(self,num):                       # 定义 setnum 方法以设置__num 属性
...     self.__num = num
...
>>> class B:                                    # 定义类 B
... nameb = 'B'                                 # 定义 nameb 属性
... __numb = 2                                  # 定义__numb 属性
... def show(self):                             # 定义 show 方法
...     print(self.nameb)
...     print(self.__numb)
... def setname(self,name):                     # 定义 setname 方法
...     self.nameb = name
...
>>> class C(A,B):                               # 通过多重继承创建 C 类
... def showall(self):                          # 定义 showall 方法
...     print(self.name)
...     print(self.nameb)
...
>>> c = C()                                     # 实例化 C 类生成 c 对象
>>> c.showall()                                 # 调用 showall 方法
A
B
>>> c.show()                                    # 调用 show 方法，此处调用的是 A 类的 show 方法
A
1
>>> c.setnum(3)                                 # 调用 setnum 方法，即 A 类的 setnum 方法
>>> c.show()
A
3
>>> c.setname('D')                              # 调用 setname 方法，即 B 类的 setname 方法
>>> c.showall()
A
D
```

如果需要在类 C 中使用类 B 的 show 方法，则可以按如下所示代码修改类 C。

```
>>> class C(A,B):                               # 通过多重继承创建 C 类
... def showall(self):
```

```
...     print(self.name)
...     print(self.nameb)
... show = B.show                                    # 这里表明 show 方法为 B 类的 show 方法
...
>>> c = C()
>>> c.show()                                         # 调用 show 方法，此处调用的是 C 类的 show 方法
B
2
```

6.5 在类中重载方法和运算符

前面介绍过，当继承某一个或几个类时，当前定义的类也继承父类的方法。重载，是指重新定义父类中的方法。在 Python 中，不仅可以重载方法，而且还可以重载运算符，例如，“+”、“-”、“*”、“/” 等，以适用所创建的类进行相应的运算。

6.5.1 方法重载

通过继承而创建的类，其父类的方法不一定能满足类的需要。新类实际上只是修改部分功能，为了避免重新命名函数，可以使用方法重载来解决。或者，新类需要重新初始化，此时就可以重载__init__方法来实现。

方法的重载实际上就是在类中使用 def 关键字重新定义父类中已有的方法。如果重载父类中的方法，但又需在类中先使用父类的该方法，则可以使用父类名＋“.”＋方法名的形式调用。例如，重载__init__方法时，若也需要使用父类的__init__方法，则可以在__init__前加上父类名来调用该方法。

下面的代码首先定义了一个父类，然后通过继承创建一个新类，并且重载父类的方法。

```
>>> class human:                                     # 定义 human 类
...     __age = 0                                    # 定义__age 属性
...     __sex = ''                                   # 定义__sex 属性
...     __height = 0                                 # 定义__height 属性
...     __weight = 0                                 # 定义__weight 属性
...     name = ''                                    # 定义 name 属性
...     def __init__(self,age,sex,height,weight):        # 重载__init__方法
...         self.__age = age                         # 初始化__age 属性
...         self.__sex = sex                         # 初始化__sex 属性
...         self.__height = height                   # 初始化__height 属性
...         self.__weight = weight                   # 初始化__weight 属性
...     def setname(self,name):                      # 定义 setname 方法
...         self.name = name
...     def show(self):                              # 定义 show 方法
...         print(self.name)
...         print(self.__age)
...         print(self.__sex)
...         print(self.__height)
...         print(self.__weight)
...
>>> class student(human):                           # 通过继承 human 类生成 student 类
```

```
... __classes = 0                                    # 定义__classes 属性
... __grade = 0                                      # 定义__grade 属性
... __num = 0                                        # 定义__num 属性
... def __init__(self,classes,grade,num,age,sex,height,weight):    # 重载
__init__方法
...     self.__classes = classes
...     self.__grade = grade
...     self.__num = num
...     human.__init__(self,age,sex,height,weight) # 调用 human 类的__init__方法,初
始化 human 类的属性
... def show(self):                          # 重载 show 方法
...     human.show(self)                     # 调用 human 类的 show 方法
...     print(self.__classes)
...     print(self.__grade)
...     print(self.__num)
...
>>> a = student(12,3,20070305,19,'male',175,65)     # 实例化生成 a 对象
>>> a.setname('Tom')                         # 调用 setname 方法
>>> a.show()                                 # 调用 show 方法，即用重载后的 show 方法输出属性
Tom
19
male
175
65
12
3
20070305
```

6.5.2 运算符重载

由于在 Python 中，运算符都有其相对应的函数。因此在 Python 中，运算符重载不需要像在 C++中那样使用 operator 关键字。在类中，运算符对应类中的一些专有方法。因此运算符的重载实际上是对运算符对应的专有方法的重载。部分运算符和类的专有方法名对照如表 6-2 所示。

表 6-2 部分运算符与类的专有方法名对照表

运算符	专有方法
+	__add__
-	__sub__
*	__mul__
/	__div__
%	__mod__
**	__pow__

```
>>> class MyList:                                          # 定义 MyList 类
... __mylist = []                                         # 定义__mylist 属性
... def __init__(self, *args):                            # 重载__init__方法
...     self.__mylist = []      # 此处相当于__mylist 初始化，避免多个实例对象数据混合
...     for arg in args:
...         self.__mylist.append(arg)
```

```
... def __add__(self,n):                                    # 重载 "+" 运算符
...     for i in range(0,len(self.__mylist)):
...         self.__mylist[i] = self.__mylist[i] + n
... def __sub__(self,n):                                    # 重载 "-" 运算符
...     for i in range(0,len(self.__mylist)):
...         self.__mylist[i] = self.__mylist[i] - n
... def __mul__(self,n):                                    # 重载 "*" 运算符
...     for i in range(0,len(self.__mylist)):
...         self.__mylist[i] = self.__mylist[i] * n
... def __div__(self,n):                                    # 重载 "/" 运算符
...     for i in range(0,len(self.__mylist)):
...         self.__mylist[i] = self.__mylist[i] / n
... def __mod__(self,n):                                    # 重载 "%" 运算符
...     for i in range(0,len(self.__mylist)):
...         self.__mylist[i] = self.__mylist[i] % n
... def __pow__(self,n):                                    # 重载 "**" 运算符
...     for i in range(0,len(self.__mylist)):
...         self.__mylist[i] = self.__mylist[i] ** n
... def __len__(self):                                      # 重载 len 方法
...     return len(self.__mylist)
... def show(self):                                         # 定义 show 方法
...     print(self.__mylist)
...
>>> l = MyList(1,2,3,4,5)                                   # 实例化生成 l 对象
>>> l.show()                                                # 调用 show 方法
[1, 2, 3, 4, 5]
>>> l + 5                                                   # 此处将调用__add__方法
>>> l.show()                                                # 调用 show 方法
[6, 7, 8, 9, 10]
>>> l - 3                                                   # 此处将调用__sub__方法
>>> l.show()                                                # 调用 show 方法
[3, 4, 5, 6, 7]
>>> l * 6                                                   # 此处将调用__mul__方法
>>> l.show()                                                # 调用 show 方法
[18, 24, 30, 36, 42]
>>> l / 3                                                   # 此处将调用__div__方法
>>> l.show()                                                # 调用 show 方法
[6, 8, 10, 12, 14]
>>> l % 3                                                   # 此处将调用__mod__方法
>>> l.show()                                                # 调用 show 方法
[0, 2, 1, 0, 2]
>>> l ** 3                                                  # 此处将调用__pow__方法
>>> l.show()                                                # 调用 show 方法
[0, 8, 1, 0, 8]
>>> len(l)                                                  # 此处将调用__len__方法
5
>>> b = MyList(2,3,5,6,7,8,9)                               # 实例化生成 b 对象
>>> len(b)                                                  # 此处将调用__len__方法
7
>>> b.show()                                                # 调用 show 方法
[2, 3, 5, 6, 7, 8, 9]
>>> b - 5                                                   # 此处将调用__sub__方法
```

```
>>> b.show()                                          # 调用 show 方法
[-3, -2, 0, 1, 2, 3, 4]
>>> l.show()                                  # 调用 l 的 show 方法，比较 l 和 b 的值
[0, 8, 1, 0, 8]
```

6.6 在模块中定义类

类与函数一样，也可以写到模块中。在其他的脚本中可以通过导入模块名来使用模块中已定义的类。模块中类的使用方式与模块中的函数类似。实际上，可以将模块中的类当作函数一样来使用。将 6.5 节中定义的 MyList 类整理一下，保存在 MyList.py 中。代码如下。

```
# -*- coding:utf-8 -*-
# file: MyList.py
#
class MyList:                                    # 定义 MyList 类
    __mylist = []                                # 定义__mylist 属性
    def __init__(self, *args):                   # 重载__init__方法
        self.__mylist = []      # 此处相当于__mylist 初始化，避免多个实例对象数据混合
        for arg in args:
            self.__mylist.append(arg)
    def __add__(self,n):                         # 重载 "+" 运算符
        for i in range(0,len(self.__mylist)):
            self.__mylist[i] = self.__mylist[i] + n
    def __sub__(self,n):                         # 重载 "-" 运算符
        for i in range(0,len(self.__mylist)):
            self.__mylist[i] = self.__mylist[i] - n
    def __mul__(self,n):                         # 重载 "*" 运算符
        for i in range(0,len(self.__mylist)):
            self.__mylist[i] = self.__mylist[i] * n
    def __div__(self,n):                         # 重载 "/" 运算符
        for i in range(0,len(self.__mylist)):
            self.__mylist[i] = self.__mylist[i] / n
    def __mod__(self,n):                         # 重载 "%" 运算符
        for i in range(0,len(self.__mylist)):
            self.__mylist[i] = self.__mylist[i] % n
    def __pow__(self,n):                         # 重载 "**" 运算符
        for i in range(0,len(self.__mylist)):
            self.__mylist[i] = self.__mylist[i] ** n
    def __len__(self):                           # 重载 len 方法
        return len(self.__mylist)
    def show(self):                              # 定义 show 方法
        print(self.__mylist)
```

然后编写一个 UseMyList.py 脚本，使用 MyList.py 中的 MyList 类。其代码如下。

```
# -*- coding:utf-8 -*-
# file: UseMyList.py
#
import MyList                                    # 导入 MyList 模块
l = MyList.MyList(1,2,3,4,5)                     # 调用 MyList 类实例化生成 l 对象
l.show()                                         # 调用 show 方法
```

```
l + 10                                          # 此处将调用__add__方法
l.show()                                        # 调用 show 方法
l * 2                                           # 此处将调用__add__方法
l.show()                                        # 调用 show 方法
print(len(l))
l ** 3                                          # 此处将调用__add__方法
l.show()                                        # 调用 show 方法
```

运行 UseMyList.py，输出如下。

```
[1, 2, 3, 4, 5]
[11, 12, 13, 14, 15]
[22, 24, 26, 28, 30]
5
[10648, 13824, 17576, 21952, 27000]
```

6.7 本章小结

面向对象是 Python 的基本特征，前面各章的例子都是采用面向过程方式编写的。本章详细介绍了 Python 中的面向对象思想，了解了类和对象的相关概念，然后介绍了在 Python 中定义和使用类的方法，接着介绍了使用类编程的相关知识，如类的属性和方法、类的继承、在类中重载方法和运算符、在模块中定义类等。

在下一章中，将介绍 Python 基本使用的另一方面：异常处理与程序调试。

◆ ◆ ◆

第 7 章　异常处理与程序调试

本章包括

- 用 try 语句捕获异常
- 多重异常的捕获
- 自定义异常类
- 在 PythonWin 中调试程序
- 常见异常的处理
- 用代码抛出异常
- 使用 pdb 调试 Python 脚本

异常通常是 Python 脚本在运行过程中引发的错误。如果在脚本中未包含对相关异常进行处理的代码，那么脚本将会由于异常而终止运行。在 Python 中，可以为脚本添加异常处理，以应对可能出现的错误，从而使脚本更“健壮”。

7.1　异常的处理

在脚本运行过程中，常见的异常有除零、下标越界等。在 Python 中，当这些异常产生时，可以捕获这些异常，并编写相关的处理语句，使脚本不会意外中断。

7.1.1　用 try 语句捕获异常

在 Python 中，可以使用 try 语句来处理异常。和 Python 中的其他语句一样，try 语句也要使用缩进结构。try 语句有一个可选的 else 语句块。一般的 try 语句形式如下。

```
try:
    <语句>                          # 要进行捕捉异常的语句
except <异常名 1>:                  # 要进行处理的异常
    <语句>                          # 对异常进行处理的语句
except <异常名 2>:                  # 要进行处理的异常
    <语句>                          # 对异常进行处理的语句
else:
    <语句>                          # 如果异常未触发，则执行该语句
```

try 语句在脚本中的执行流程如图 7-1 所示。

try 语句还有一种不包含 except 和 else 语句的特殊形式，其形式如下。

```
try:
    <语句>
finally:
    <语句>
```



图 7-1　try 语句执行流程

不管 try 语句块中是否发生异常，都执行 finally 语句块。下面是使用 try 语句处理异常的代码示例。

```
>>> l = [1,2,3]                                    # 定义一个列表
>>> l[5]                                           # 此处列表越界导致错误
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
IndexError: list index out of range
>>> try:                                           # 使用 try 语句捕获异常
... l[5]                                           # 引发列表越界错误
... except:                                        # 此处未使用异常名，表示捕获所有异常
... print('Error')                                 # 如果引发异常则打印 "Error"
... else:                                          # 如果没有异常则将执行 else 语句块中的内容
... print('No Error')
...
Error                                              # 输出 Error 表示引发了异常
>>> try:
... l[2]                                           # 此处列表没有越界，将不会引发异常
... except:
... print('Error')
... else:
... print('No Error')
...
3
No Error                                           # 此处为 else 语句块中的内容，表示未引发异常
>>> try:
... l[2]/0
... except IndexError:                             # 只捕获 IndexError 异常，也就是列表越界异常
... print('Error')
... else:
... print('No Error')
...
# 脚本运行出错，因为在脚本中只捕获 IndexError 而没有捕获 ZeroDivisionError，也就是除零异常
```

```
Traceback (most recent call last):
  File "<interactive input>", line 2, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> try:                                    # 捕获多个异常，并分别处理
... l[2]/0
... except IndexError:                      # 捕获 IndexError 异常
... print( 'IndexError')
... except ZeroDivisionError:               # 捕获 ZeroDivisionError 异常
... print( 'ZeroDivisionError')
... else:
... print('No Error')
...
ZeroDivisionError
>>> try:                                    # 使用 try…finally 语句
... l[2]                                    # 此处不会引发异常
... finally:
... print('A')                              # 仅打印字母 "A"，表示语句执行
...
3
A
>>> try:
... l[5]                                    # 此处引发异常
... finally:
... print('A')
...
A                                           # 虽然引发异常，但 finally 语句块仍被执行
Traceback (most recent call last):
  File "<interactive input>", line 2, in <module>
IndexError: list index out of range
```

7.1.2 常见异常的处理

在 except 中可以捕获指定的异常，除此之外，except 语句还可以捕获产生异常时的附加数据。Python 中常用的内置异常如表 7-1 所示。

表 7-1 Python 中常用的内置异常

异常名	描述
AttributeError	调用不存在的方法引发的异常
EOFError	遇到文件末尾引发的异常
ImportError	导入模块出错引发的异常
IndexError	列表越界引发的异常
IOError	I/O 操作引发的异常，如打开文件出错等
KeyError	使用字典中不存在的关键字引发的异常
NameError	使用不存在的变量名引发的异常
TabError	语句块缩进不正确引发的异常
ValueError	搜索列表中不存在的值引发的异常
ZeroDivisionError	除数为零引发的异常

except 语句主要有以下几种用法。

```
except:                                          # 捕获所有异常
except <异常名>:                                  # 捕获指定异常
except (异常名1,异常名2):                         # 捕获异常名1或异常名2
except <异常名> as <数据>:                        # 捕获指定异常及其附加的数据
except (异常名1,异常名2) as <数据>:               # 捕获异常名1或异常名2及异常的附加数据
```

以下代码是使用 except 捕获异常。

```
>>> l = [1,2,3]                                  # 定义一个列表
>>> try:                                         # 异常处理
... l[5]
... except IndexError as Error:                  # 捕获 IndexError 异常并获取其附加数据
... print(Error )                                # 打印异常的附加数据
... else:
... print('No Error')                            # 如果未触发异常，则打印 "No Error"
...
list index out of range                          # 此处为异常的附加数据
>>> try:
... l[2]/0
... except (IndexError,ZeroDivisionError): # 捕获 IndexError 异常或
ZeroDivisionError 异常
... print('Error')
... else:
... print('No Error')
...
Error
>>> try:
... l[5]/0
... except:                                      # 捕获所有异常
... print('Error')
... else:
... print('No Error')
...
Error
>>> try:
... l[2]/0
... except (IndexError,ZeroDivisionError) as value:
                          # 捕获 IndexError 异常或 ZeroDivisionError 异常及其附加数据
... print(value)
... else:
... print('No Error')
...
division by zero
```

7.1.3 多重异常的捕获

在 Python 中，可以在 try 语句中嵌套另外一个 try 语句。由于 Python 将 try 语句放在堆栈中，因此一旦引发异常，Python 将匹配最近的 except 语句。如果 except 能够处理该异常，则外围的 try 语句将不会捕获异常。如果 except 忽略该异常，则异常将被外围 try 语句捕获。

下面的代码是在 try 语句中嵌套另外一个 try 语句。

```
>>> l = [1,2]                                    # 定义一个列表
```

```
>>> try:                                # 嵌套 try 语句
... try:
...     l[5]
... except:                             # 捕获所有异常
...     print('Error1')                 # 打印 "Error1"
... except:                             # 捕获所有异常
... print('Error2')                     # 打印 "Error2"
... else:
... print('No Error')
...
Error1                                  # 此处为内嵌 try 语句输出
No Error                                # 此处为外围 try 语句输出,表示 try 语句块中未引发异常
>>> try:
... try:
...     l[1]/0
... except IndexError:                  # 仅捕获 IndexError 异常
...     print('Error1')
... except:                             # 捕获所有异常
... print('Error2')
... else:
... print('No Error')
...
Error2                                  # 异常被外围 try 语句捕获
>>> try:
... try:
...     l[1]/'s'
... except IndexError:                  # 仅捕获 IndexError 异常
...     print('Error1')
... except ZeroDivisionError:           # 仅捕获 ZeroDivisionError 异常
... print('Error2')
... else:
... print('No Error')
...
Traceback (most recent call last): # 最终脚本执行出错
  File "<interactive input>", line 3, in <module>
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

7.2 用代码抛出异常

除了 Python 内置的异常外，在脚本中还可以通过使用 raise 语句手动引发异常。在类中也可以使用 raise 引发异常，并向异常传递数据。

使用 raise 可以定义新的异常类型，以适应脚本的需要。例如，对用户输入数据的长度有要求，则可以使用 raise 引发异常，以确保数据输入的长度符合要求。

7.2.1 用 raise 抛出异常

使用 raise 引发异常十分简单。raise 有以下几种使用方式。

```
raise  异常名
raise     异常名,附加数据
```

```
raise  类名
```

下面的代码是使用 try 语句捕获由 raise 引发的异常。

```
>>> raise Exception                          # 使用 raise 引发异常
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
Exception
>>> try:                                     # 使用 try 语句捕获 raise 引发的异常
...     raise Exception
...     except Exception:                    # 捕获 "Exception" 异常
...     print('Error')
...     else:
...     print('No Error')
...
Error
>>> try:
...     raise Exception('Raise an Exception')  # 使用 raise 引发异常并传递附加数据
...     except Exception as data:              # 捕获异常和附加数据
...     print(data)
...     else:
...     print('No Error')
...
Raise an Exception
>>> def fun(n):                              # 定义一个函数
...     if n == 0:
...         raise ZeroDivisionError('n is zero')   # 在函数中使用 raise 引发异常
...     else:
...         print(n)
...
>>> try:                                     # 使用 try 语句捕获函数中的异常
...     fun(0)
...     except ZeroDivisionError as data:
...     print(data)
...
n is zero
>>> class A(Exception):                      # 定义一个类，继承自 Exception 或其子类
...     def show(self):                      # 定义类的 show 方法
...         print('A')
...
>>> try:
...     raise A                              # 使用 raise 引发一个类异常
...     except A:
...     print('Error')
...     else:
...     print('No Error')
...
Error
```

7.2.2 assert——简化的 raise 语句

在 Python 中，使用 assert 语句同样可以引发异常。但与 raise 语句不同，assert 语句是在条件测试为假时，才引发异常。assert 语句的一般形式如下。

```
assert  <条件测试>,<异常附加数据>                    # 其中异常附加数据是可选的
```

下面的代码是使用 assert 语句引发异常。

```
>>> l = []                                          # 定义一个列表
>>> assert len(l)                                   # 如果列表为空，则使用 assert 引发异常
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AssertionError
>>> assert len(l),'Error'                           # 向异常传递附加数据
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AssertionError: Error
>>> try :                                           # 使用 try 语句捕获 assert 异常
... assert len(l),'Error'
... except:                                         # 捕获所有异常
... print('Error')
... else:
... print('No Error')
...
Error
>>> l.append(1)                                     # 向列表中添加成员
>>> assert len(l)                                   # 此时列表不为空，assert 将不会引发异常
```

从上述代码可以看出，assert 相当于 raise 语句和 if 语句联合使用。例如，下面的 assert 语句：

```
assert len(l)
```

可以改写为下面的 raise 语句：

```
if __debug__:
    if len(l):
        raise AssertionError,<附加数据>
```

需要注意的是，assert 语句一般用于开发时对程序条件的验证，只有当内置__debug__为 True 时，assert 语句才有效。当 Python 脚本以-O 选项编译成为字节码文件时，assert 语句将被移除。

7.2.3 自定义异常类

在 Python 中，可以通过继承 Exception 类来创建自己的异常类。异常类和其他的类并没有区别，一般在异常类中仅需要定义几个属性信息。

下面的代码是通过继承 Exception 类来生成一个 MyError 类。

```
>>> class MyError(Exception):                       # 通过继承 Exception 类来生成 MyError 类
... def __init__(self,data):                        # 重载__init__方法
...     self.data = data
... def __str__(self):                              # 重载__str__方法
...     return self.data
...
>>> raise MyError('Error')                          # 使用 raise 引发 MyError 异常
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
```

```
MyError: Error
>>> try:                                    # 使用 try 语句捕获 MyError 异常
... raise MyError('Raise MyError')
... except MyError as data:                 # 捕获异常和附加数据
... print(data)
... else:
... print('No error')
...
Raise MyError
```

7.3 使用 pdb 调试 Python 脚本

在 Python 中，脚本的语法错误可以被 Python 解释器发现，但脚本逻辑上的错误，或者其他的一些变量使用错误却不容易被发现。如果脚本运行后没有获得预想的结果，则需要对脚本进行调试。

pdb 模块是 Python 自带的调试模块。使用 pdb 模块可以为脚本设置断点、单步执行、查看变量值等。pdb 模块可以用命令行参数的形式启动，也可以通过 import 将其导入使用。

通过 import 导入 pdb 模块后，就可以使用 pdb 模块的函数对脚本进行调试。常用的 pdb 模块的函数可以分为以下几类。

7.3.1 运行语句

在 Python 中，可以使用 pdb 模块的 run 函数来调试语句块。其参数原型如下。

```
run( statement[, globals[, locals]])
```

其参数含义如下。

- ◆ statement　　要调试的语句块，以字符串的形式表示。
- ◆ globals　　可选参数，设置 statement 运行的全局环境变量。
- ◆ locals　　可选参数，设置 statement 运行的局部环境变量。

以下代码是使用 run 函数调试语句块。

```
>>> import pdb                          # 导入 pdb 模块
>>> pdb.run('''                         # 调用 run 函数执行一个 for 语句
... for i in range(0,3):
...     i = i ** 2
...     print(i)
... ''')
> <string>(2)<module>()->None
(Pdb) n                                 # "(Pdb)" 为调试命令提示符，表示可以输入调试命令
> <string>(3)<module>()->None
(Pdb) n                                 # 执行下一行
> <string>(4)<module>()->None
(Pdb) print(i)                          # print 打印变量 i 的值
0
(Pdb) continue                          # 继续运行程序
0
1
4
```

7.3.2 运行表达式

在 Python 中，可以使用 pdb 模块的 runeval 函数来调试表达式。其参数原型如下。

```
runeval( expression[, globals[, locals]])
```

其参数含义如下。

- expression　　要调试的表达式，以字符串的形式。
- globals　　可选参数，设置 statement 运行的全局环境变量。
- locals　　可选参数，设置 statement 运行的局部环境变量。

以下代码是使用 runeval 函数来调试表达式。

```
>>> import pdb                                  # 导入 pdb 模块
>>> l = [1,2,3]                                 # 定义一个列表
>>> pdb.runeval('l[1]')                         # 使用 runeval 调试表达式 l[1]
> <string>(1)<module>()->2
(Pdb) n                                         # 进入调试状态，使用 n 命令，单步执行
--Return--
> <string>(1)<module>()->2
(Pdb) n                                         # 使用 n 命令，单步执行
2                                               # 表达式的值
>>> pdb.runeval('3+5*6/2')                      # 使用 runeval 调试表达式 3+5*6/2
> <string>(1)<module>()->2
(Pdb) n                                         # 进入调试状态，使用 n 命令，单步执行
--Return--
> <string>(1)<module>()->18
(Pdb) n                                         # 使用 n 命令，单步执行
18                                              # 表达式的值
```

7.3.3 运行函数

在 Python 中，可以使用 pdb 模块的 runcall 函数来调试函数。其函数原型如下。

```
runcall( function[, argument, ...])
```

其参数含义如下。

- function　　函数名。
- argument　　函数的参数。

以下代码是使用 runcall 函数来调试函数。

```
>>> import pdb                                  # 导入 pdb 模块
>>> def sum(*args):                             # 定义函数 sum，求所有参数之和
...     r = 0
...     for arg in args:
...             r = r + arg
...     return r
```

```
...
>>> pdb.runcall(sum,1,2,3,4)                    # 使用 runcall 调试函数 sum
> <stdin>(2)sum()
(Pdb) n                                         # 进入调试状态，使用 n 命令，单步执行
> <stdin>(3)sum()
(Pdb) n                                         # 使用 n 命令，单步执行
> <stdin>(4)sum()
(Pdb) print(r)                                  # 使用 print 打印变量 r 的值
0
(Pdb) continue                                  # 使用 continue 继续执行
10                                              # 函数返回值
>>> pdb.runcall(sum,1,2,3,4,5,6)                # 使用 runcall 调试函数 sum
> <stdin>(2)sum()
(Pdb) continue                                  # 使用 continue 继续执行
21                                              # 函数返回值
```

7.3.4 设置硬断点

在 Python 中，可以使用 pdb 模块的 set_trace 函数在脚本中设置硬断点。set_trace 函数一般在“.py”脚本中使用。其函数原型如下。

```
set_trace()
```

以下代码是使用 set-trace 函数在脚本中设置硬断点。

```
# -*- coding:utf-8 -*-
# file: debug.py
#
import pdb                                      # 导入 pdb 模块
pdb.set_trace()                                 # 使用 set_trace 函数设置硬断点
for i in range(0,5):
    i = i * 5
    print(i)
```

运行脚本后显示如下。

```
> e:\[ython\第 7 章\debug.py(6)<module>()
-> for i in range(0,5):
(Pdb) list                                      # 使用 list 列出脚本内容
  1     # -*- coding:utf-8 -*-
  2     # file: debug.py
  3     #
  4     import pdb
  5     pdb.set_trace()                         # 使用 set_trace 函数设置硬断点
  6  -> for i in range(0,5):
  7           i = i * 5
  8           print(i)
[EOF]
(Pdb) continue                                  # 使用 continue 继续执行
0
5
10
15
20
```

7.3.5 pdb 调试命令

pdb 中的调试命令可以完成单步执行、打印变量值、设置断点等功能。在前面几节中已经使用了部分 pdb 的调试命令与被调试的脚本进行交互。pdb 的部分调试命令如表 7-2 所示。

表 7-2 pdb 的部分调试命令

完整命令	简写命令	描述
args	a	打印当前函数的参数
break	b	设置断点
clear	cl	清除断点
condition	无	设置条件断点
continue	c 或者 cont	继续运行，直到遇到断点或者脚本结束
disable	无	禁用断点
enable	无	启用断点
help	h	查看 pdb 帮助
ignore	无	忽略断点
jump	j	跳转到指定行数运行
list	l	列出脚本清单
next	n	执行下条语句，遇到函数不进入其内部
p	p	打印变量值，也可以用 print
quit	q	退出 pdb
return	r	一直运行到函数返回
tbreak	无	设置临时断点，断点只中断一次
step	s	执行下一条语句，遇到函数进入其内部
where	w	查看所在的位置
!	无	在 pdb 中执行语句

以下代码是通过命令行启动 pdb 对脚本进行调试。在 Windows 命令行中进入脚本所在的目录，输入以下命令，启动 Python。

```
E:\python\第 7 章>python -m pdb prime.py
```

脚本运行后输入以下命令。

```
> e:\python\第 7 章\prime.py(4)<module>()
-> import math
(Pdb) list                              # 先停在这里，前边有 "(Pdb)" 提示符，输入 list 命令
  1     # -*- coding:utf-8 -*-          # list 命令默认只列出前 11 行
  2     # file: prime.py
  3     #
  4  -> import math
  5     # isprime 函数判断一个整数是否为素数。
  6     # 如果 i 能被 2 到 i 的平方根内的任意一个数整除，
  7     # 则 i 不是素数，返回 0，否则 i 是素数，返回 1。
  8     def isprime(i):
```

```
 9           for t in range( 2, int(math.sqrt(i)) + 1 ):
10                 if i % t == 0:
11                       return 0
(Pdb) l 14,17                          # 使用 list 命令列出 14 到 17 行的脚本内容
14     print('100~110 之间的素数有: ')
15     for i in range(100,110):
16           if isprime(i):
17                 print(i)
(Pdb) b 14                             # 使用 b 在第 14 行设置断点
Breakpoint 1 at e: \python\第 7 章\prime.py:14        # 返回断点编号 1
(Pdb) b isprime                        # 在函数 isprime 上设置断点
Breakpoint 2 at e:\python\第 7 章\prime.py:8          # 返回断点编号 2
(Pdb) c                                # 使用 c 命令运行脚本
> e:\ python\第 7 章\prime.py(14)<module>()           # 停在断点 1 处，即第 14 行
-> print('100~110 之间的素数有: ')
(Pdb) c                                # 使用 c 命令继续运行脚本
100~200 之间的素数有:                  # 第 14 行脚本输出
> e:\python\第 7 章\prime.py(9)isprime()              # 停在断点 2 处，即 isprime 函数处
-> for t in range( 2, int(math.sqrt(i)) + 1 ):
(Pdb) b 15                             # 在第 15 行设置断点
Breakpoint 3 at e:\python\第 7 章\prime.py:15         # 返回断点编号 3
(Pdb) disable 2                        # 禁用断点 2，即 isprime 函数处的断点
(Pdb) c                                # 继续运行脚本
> e:\python\第 7 章\prime.py(15)<module>()            # 停在断点 3 处，即第 15 行
-> for i in range(100,110):
(Pdb) print(i)                         # 使用 print 打印变量 i 的值
100
(Pdb) c                                # 继续运行脚本
101
> e:\python\第 7 章\prime.py(15)<module>()            # 停在断点 3 处，即第 15 行
-> for i in range(100,110):
(Pdb) p i                              # 使用 p 打印变量 i 的值
101
(Pdb) enable 2                         # 恢复断点 2，即 isprime 函数处的断点
(Pdb) c                                # 继续运行脚本
> e:\python\第 7 章\prime.py(9)isprime()      # 停在断点 2 处，即 isprime 函数处的断点
-> for t in range( 2, int(math.sqrt(i)) + 1 ):
(Pdb) n                                # 单步执行下一语句
> e:\python\第 7 章\prime.py(10)isprime()
-> if i % t == 0:                      # 停在下一语句处
(Pdb) print(t)                         # 使用 print 打印变量 t 的值
2
(Pdb) cl                               # 清除所有断点，输入 y 确认
Clear all breaks? y
(Pdb) c                                # 继续运行脚本
103
105
107
109
The program finished and will be restarted          # 脚本运行结束，回到开始处
```

```
> e:\python\第 7 章\prime.py(4)<module>()
-> import math
(Pdb) q                                    # 使用 q 命令退出 pdb
```

7.4 在 PythonWin 中调试程序

在 PythonWin 中集成了 Python 脚本的调试环境，可以方便地对脚本进行调试。如果觉得使用 pdb 的调试命令过于烦琐的话，则可以使用 PythonWin 的脚本调试功能。

运行 PythonWin 后单击【View】命令。选择【View】菜单下的【Toolbars】下拉菜单，单击【Debugging】命令，将其前边打钩，如图 7-2 所示。将会出现如图 7-3 所示的调试工具栏。



图 7-2 选中【Debugging】命令



图 7-3 PythonWin 中的调试工具栏

其中，各按钮说明如下。

- 查看变量或者表达式的值。
- 查看堆栈。
- 查看断点列表。
- 设置断点。
- 清除断点。
- 单步执行，相当于 step。
- 单步执行，相当于 next。
- 执行到返回，相当于 return。
- 执行脚本。
- 退出调试。

下面仍然以调试 7.3 节中的 prime.py 脚本为例，演示在 PythonWin 中调试脚本的过程，具体操作步骤如下。

step 1 在 PythonWin 中打开 prime.py 脚本。

step 2 将光标移动到第 9 行，单击设置断点按钮，在第 9 行设置断点。

step 3 将光标移动到第 14 行，单击设置断点按钮，在第 14 行设置断点。断点设置完成后，如图 7-4 所示。

step 4 单击运行脚本按钮▶执行脚本，此时脚本中断在第 14 行，如图 7-5 所示。



图 7-4 在 PythonWin 中设置断点



图 7-5 脚本中断在第 14 行

step 5 单击查看按钮，将弹出如图 7-6 所示的浮动窗口。

step 6 双击浮动窗口中的 "New Item"，输入 i，添加查看变量 i 的值。

step 7 单击运行脚本按钮▶，执行脚本，脚本将中断在第 9 行，如图 7-7 所示。可以看到浮动窗口中 i 的值为 100。



图 7-6 PythonWin 中的浮动窗口



图 7-7 脚本中断在第 9 行

step 8 单击显示断点列表按钮，打开如图 7-8 所示的断点列表窗口。

step 9 单击清除断点按钮，将清除所有断点，如图 7-9 所示。



图 7-8 断点列表窗口



图 7-9 清除所有断点

step 10 单击退出调试按钮，将退出调试状态，如图 7-10 所示。



图 7-10 退出调试

7.5 本章小结

本章介绍了两个方面的知识点，异常的处理和程序的调试。首先介绍了异常的基本处理方法，以及用 try 语句捕获异常、常见异常的处理、多重异常的捕获等内容。接着介绍了用代码抛出异常的方法和自定义异常类的方法。最后介绍了两种调试脚本的方法：使用 pdb 调试 Python 脚本程序和使用 PythonWin 调试程序。

在下一章中，将介绍 Python 在多媒体编程方面的内容。

◆ ◆ ◆

第 8 章　Python 多媒体编程

本章包括

- 安装 PyOpenGL
- 使用 PyOpenGL 绘制文字、二维图形、三维图形
- 安装 PyGame
- 使用 PyOpenGL 创建窗口使用
- 使用 PyOpenGL 播放音频文件
- 使用 PyGame 编写简单的游戏

通过 Python 的扩展模块，可以使用 Python 创建三维图形、播放音乐等多媒体编程。使用 PyOpenGL 模块可以创建三维图形。在 Windows 下，可以使用 DirectSound 和 WMPlayer.OCX 来播放音频文件。另外使用 PyGame 模块可以编写游戏。

8.1　使用 PyOpenGL 绘制三维图形

PyOpenGL 模块是对 OpenGL 的封装，OpenGL（Open Graphics Library）提供了不同的函数调用，可以绘制简单的图形及复杂的三维图形。使用 PyOpenGL 模块可以调用 OpenGL 中的函数绘制图形。

8.1.1　安装 PyOpenGL

由于 PyOpenGL 没有提供 Python 3 的安装程序，因此本章以 Python 2.5+PyOpenGL 为例，介绍 PyOpenGL 的使用方法。等 PyOpenGL 发布官方的 Python 3 安装程序后，再安装到 Python 3 中即可同样使用。

PyOpenGL 的安装程序发布在 http://pyopengl.sourceforge.net/网站。另外，在 Python 官方网站的 PyPi 中也提供下载，下载页面是 https://pypi.python.org/pypi/PyOpenGL/3.0.1，下载 Windows 下的安装程序 PyOpenGL-3.0.1.win32.exe，下载完成后，就可以按步骤进行安装（首先应安装 Python 2.5）。

step 1　双击安装程序 PyOpenGL-3.0.1.win32.exe，打开如图 8-1 所示的【安装向导】界面。

step 2　单击【下一步】按钮，显示如图 8-2 所示界面，选择 Python 的版本，注意这里选择的是 Python 2.5，若选择 Python 3.2，安装后将无法使用 PyOpenGL。

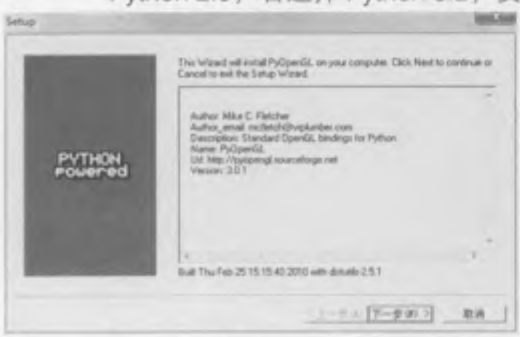


图 8-1　【安装向导】界面

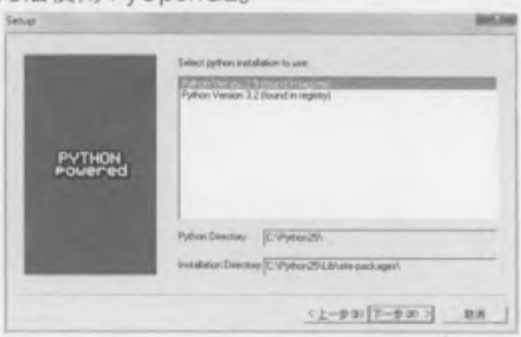


图 8-2　选择 Python 版本

step 3 单击【下一步】按钮，显示如图 8-3 所示界面，确认安装。

step 4 单击【下一步】按钮，开始进行安装，经过一段时间之后安装完成，将显示如图 8-4 所示界面，单击【完成】按钮，完成 PyOpenGL 的安装。



图 8-3 【确认安装】界面



图 8-4 【安装完成】界面

8.1.2 使用 PyOpenGL 创建窗口

安装好 PyOpenGL 之后，就可以开始使用 PyOpenGL 的相关模块来编写脚本了。

使用 PyOpenGL 和使用 OpenGL 创建程序的过程基本类似，在程序中首先要对 OpenGL 进行初始化，设置相关的参数。

使用 OpenGL 程序时首先要调用 glutInit 函数，向其传递命令行参数。然后调用 glutInitDisplayMode 函数设置显示模式，调用 glutCreateWindow 函数创建窗口，调用 glutDisplayFunc 设置场景绘制函数。最后调用自定义的初始函数完成 OpenGL 的初始化，进入消息循环。

下面所示的 pyOpenGLWindow.py 脚本仅创建了一个窗口。

```
# -*- coding:utf-8 -*-
# file: pyOpenGLWindow.py
#
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import sys
class OpenGLWindow:
    def __init__(self, width = 640, height = 480, title = 'PyOpenGL'):
    # 初始化
        glutInit(sys.argv)                                          # 传递命令行参数
        glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH)  # 设置显示模式
        glutInitWindowSize(width, height)                           # 设置窗口大小
        self.window = glutCreateWindow(title)                       # 创建窗口
        glutDisplayFunc(self.Draw)                                  # 设置场景绘制函数
        self.InitGL(width, height)                                  # 调用 OpenGL 初始化函数
    def Draw(self):                                                 # 绘制场景
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT) # 清除屏幕和深度缓存
        glLoadIdentity()                                            # 重置观察矩阵
        glutSwapBuffers()                                           # 交换缓存
    def InitGL(self, width, height):                                # OpenGL 初始化函数
```

```
        glClearColor(0.0, 0.0, 0.0, 0.0)                          # 设为黑色背景
        glClearDepth(1.0)                                         # 设置深度缓存
        glDepthFunc(GL_LESS)                                      # 设置深度测试类型
        glEnable(GL_DEPTH_TEST)                                   # 允许深度测试
        glShadeModel(GL_SMOOTH)                                   # 启动平滑阴影
        glMatrixMode(GL_PROJECTION)                               # 设置观察矩阵
        glLoadIdentity()                                          # 重置观察矩阵
        gluPerspective(45.0, float(width)/float(height), 0.1, 100.0)
                                                                  # 计算屏幕高宽比
        glMatrixMode(GL_MODELVIEW)                                # 设置观察矩阵
    def MainLoop(self):                                           # 进入消息循环
        glutMainLoop()
window = OpenGLWindow()                                           # 创建窗口
window.MainLoop()                                                 # 进入消息循环
```

运行 pyOpenGLWindow.py 后，将创建如图 8-5 所示的窗口。



图 8-5　PyOpenGL 窗口



需要使用 C:\Python25 目录中的 Python.exe 来运行 pyOpenGLWindow.py 脚本，本节后面有关 PyOpenGL 的例子都是这样设置的。例如，可用以下命令来运行 pyOpenGLWindow.py。

```
E:\Python\第 8 章>c:\python25\python.exe pyOpenGLWindow.py
```

8.1.3　绘制文字

使用 PyOpenGL 绘制文字相对比较复杂，需要使用 glutBitmapCharacter 函数，其原型如下。

```
glutBitmapCharacter(font , character)
```

其参数含义如下。

- font　　所使用的字体。
- character　　输出字符的 ASCII 值。

可以选用的字体有以下几种。

- GLUT_BITMAP_8_BY_13
- GLUT_BITMAP_9_BY_15
- GLUT_BITMAP_TIMES_ROMAN_10
- GLUT_BITMAP_TIMES_ROMAN_24
- GLUT_BITMAP_HELVETICA_10
- GLUT_BITMAP_HELVETICA_12
- GLUT_BITMAP_HELVETICA_18

由于 glutBitmapCharacter 每次只能输出一个字符，因此需要在脚本中编写函数处理字符串。下面所示的 pyOpenGLText.py 使用 glutBitmapCharacter 在窗口中绘制文字。

```
# -*- coding:utf-8 -*-
# file: pyOpenGLText.py
#
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import sys
class OpenGLWindow:
    def __init__(self, width = 640, height = 480, title = 'PyOpenGL'):
    # 初始化
        glutInit(sys.argv)                                          # 传递命令行参数
        glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH)  # 设置显示模式
        glutInitWindowSize(width, height)                           # 设置窗口大小
        self.window = glutCreateWindow(title)                       # 创建窗口
        glutDisplayFunc(self.Draw)                                  # 设置场景绘制函数
        self.InitGL(width, height)                                  # 调用 OpenGL 初始化函数
    def Draw(self):                                                 # 绘制场景
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)          # 清除屏幕和深度缓存
        glLoadIdentity()                                            # 重置观察矩阵
        glTranslatef(0.0, 0.0, -1.0)                                # 移动位置
        glColor3f(0.0, 1.0, 0.0)                                    # 设置颜色为绿色
        glRasterPos2f(0.0, 0.0)                                     # 定位文字
        self.DrawText('PyOpenGL')                                   # 绘制文字
        glutSwapBuffers()                                           # 交换缓存
    def DrawText(self, string):                                     # 绘制文字函数
        for c in string:                                            # 循环处理字符串
            glutBitmapCharacter(GLUT_BITMAP_8_BY_13, ord(c))    # 输出文字
    def InitGL(self, width, height):                                # OpenGL 初始化函数
        glClearColor(0.0, 0.0, 0.0, 0.0)                            # 设为黑色背景
        glClearDepth(1.0)                                           # 设置深度缓存
        glDepthFunc(GL_LESS)                                        # 设置深度测试类型
        glEnable(GL_DEPTH_TEST)                                     # 允许深度测试
        glShadeModel(GL_SMOOTH)                                     # 启动平滑阴影
        glMatrixMode(GL_PROJECTION)                                 # 设置观察矩阵
```

```
        glLoadIdentity()                                          # 重置观察矩阵
        gluPerspective(45.0, float(width)/float(height), 0.1, 100.0)
                                                                  # 计算屏幕高宽比
        glMatrixMode(GL_MODELVIEW)                                # 设置观察矩阵
    def MainLoop(self):                                           # 进入消息循环
        glutMainLoop()
window = OpenGLWindow()                                           # 创建窗口
window.MainLoop()                                                 # 进入消息循环
```

运行 glutBitmapCharacter 脚本后，将新建一个窗口，并在窗口中绘制文字“PyOpenGL”，如图 8-6 所示。



图 8-6 绘制文字

8.1.4 绘制二维图形

在 PyOpenGL 中绘制图形时应以 glBegin 函数开始，当绘制完成后应调用 glEnd 函数。glBegin 函数原型如下。

```
glBegin (mode)
```

其参数含义如下。

- mode　　绘制的图形。

其中，可以选择的图形有以下几种。

- GL_POINTS　　绘制点。
- GL_LINES　　绘制直线。
- GL_LINE_STRIP　　绘制连续直线，不封闭。
- GL_LINE_LOOP　　绘制连续直线，封闭。
- GL_TRIANGLES　　绘制三角形。
- GL_TRIANGLE_STRIP　　绘制三角形串。
- GL_TRIANGLE_FAN　　绘制三角扇形。

- GL_QUADS　　绘制四边形。
- GL_QUAD_STRIP　　绘制四边形串。
- GL_POLYGON　　绘制多边形。

由于在 PyOpenGL 中没有提供直接绘制圆形的函数，因此可以使用多边形来模拟绘制圆形。下面所示的 pyOpenGLDraw2D.py 脚本绘制了几种简单的二维图形。

```
# -*- coding:utf-8 -*-
# file: pyOpenGLDraw2D.py
#
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import sys
import math
class OpenGLWindow:
    def __init__(self, width = 640, height = 480, title = 'PyOpenGL'):
    # 初始化
        glutInit(sys.argv)                                          # 传递命令行参数
        glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH)  # 设置显示模式
        glutInitWindowSize(width, height)                           # 设置窗口大小
        self.window = glutCreateWindow(title)                       # 创建窗口
        glutDisplayFunc(self.Draw)                                  # 设置场景绘制函数
        self.InitGL(width, height)                                  # 调用 OpenGL 初始化函数
    def Draw(self):                                                 # 绘制场景
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        glLoadIdentity()                                            # 重置观察矩阵
        glTranslatef(-2.0, 2.0, -6.0)                               # 移动位置
        glBegin(GL_LINES)                                           # 绘制直线
        glVertex3f(0.0, 0.0, 0.0)                                   # 直线第一点坐标
        glVertex3f(2.0, 0.0, 0.0)                                   # 直线第二点坐标
        glEnd()                                                     # 结束绘制
        glTranslatef(3.0, 0.0, 0.0)                                 # 移动位置
        glBegin(GL_POLYGON)                                         # 通过绘制多边形来模拟圆
        i = 0
        while( i <= 3.14 *2 ):
            x = 0.5 * math.cos(i)
            y = 0.5 * math.sin(i)
            glVertex3f(x, y, 0.0)
            i = i + 0.01
        glEnd()
        glTranslatef(-2, -3.0, 0.0)                                 # 移动位置
        glBegin(GL_POLYGON)                                         # 绘制三角形
        glVertex3f(0.0, 1.0, 0.0)
        glVertex3f(1.0, -1.0, 0.0)
        glVertex3f(-1.0, -1.0, 0.0)
        glEnd()
        glTranslatef(2.5, 0.0, 0.0)                                 # 移动位置
        glBegin(GL_QUADS)                                           # 绘制四边形
        glVertex3f(-1.0, 1.0, 0.0)
        glVertex3f(1.0, 1.0, 0.0)
```

```
        glVertex3f(1.0, -1.0, 0.0)
        glVertex3f(-1.0, -1.0, 0.0)
        glEnd()
        glutSwapBuffers()                                    # 交换缓存
    def InitGL(self, width, height):                         # OpenGL 初始化函数
        glClearColor(0.0, 0.0, 0.0, 0.0)                     # 设为黑色背景
        glClearDepth(1.0)                                    # 设置深度缓存
        glDepthFunc(GL_LESS)                                 # 设置深度测试类型
        glEnable(GL_DEPTH_TEST)                              # 允许深度测试
        glShadeModel(GL_SMOOTH)                              # 启动平滑阴影
        glMatrixMode(GL_PROJECTION)                          # 设置观察矩阵
        glLoadIdentity()                                     # 重置观察矩阵
        gluPerspective(45.0, float(width)/float(height), 0.1, 100.0)# 计算屏幕高宽比
        glMatrixMode(GL_MODELVIEW)                           # 设置观察矩阵
    def MainLoop(self):                                      # 进入消息循环
        glutMainLoop()
window = OpenGLWindow()                                      # 创建窗口
window.MainLoop()                                            # 进入消息循环
```

运行 pyOpenGLDraw2D.py 脚本后，将创建如图 8-7 所示的窗口，并在窗口中绘制了几种简单的几何图形。

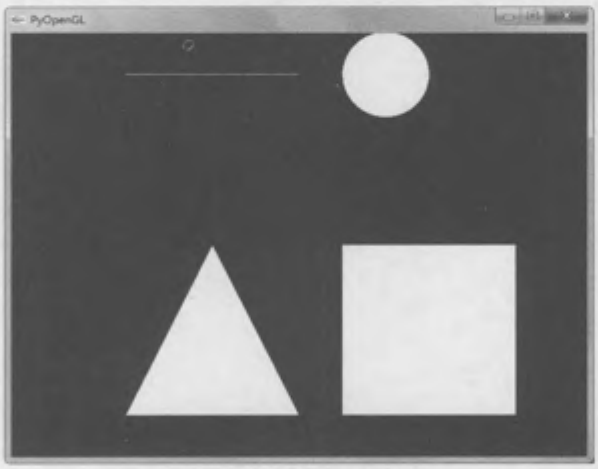


图 8-7　绘制二维图形

8.1.5 绘制三维图形

在 PyOpenGL 中还可以绘制三维图形，绘制三维图形和绘制二维图形类似，是通过设置 Z 坐标来设置图形所在的空间位置。下面的代码是用 pyOpenGLDraw3D.py 绘制一个立方体。

```
# -*- coding:utf-8 -*-
# file: pyOpenGLDraw3D.py
#
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import sys
class OpenGLWindow:
```

```
def __init__(self, width = 640, height = 480, title = 'PyOpenGL'):
# 初始化
    glutInit(sys.argv)                                          # 传递命令行参数
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH)  # 设置显示模式
    glutInitWindowSize(width, height)                           # 设置窗口大小
    self.window = glutCreateWindow(title)                       # 创建窗口
    glutDisplayFunc(self.Draw)                                  # 设置场景绘制函数
    self.InitGL(width, height)                                  # 调用 OpenGL 初始化函数
def Draw(self):                                                 # 绘制场景
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)            # 清除屏幕
    glLoadIdentity()                                            # 重置观察矩阵
    glTranslatef(1.5,0.0,-7.0)                                  # 移动位置
    glRotatef(45,1.0,1.0,1.0)                                   # 绕 X, Y, Z 轴旋转 45 度
    glBegin(GL_QUADS)                                           # 开始绘制立方体
    glColor3f(1.0,0.0,0.0)                                      # 设置颜色为红色
    glVertex3f( 1.0, 1.0,-1.0)                                  # 绘制立方体的顶面
    glVertex3f(-1.0, 1.0,-1.0)
    glVertex3f(-1.0, 1.0, 1.0)
    glVertex3f( 1.0, 1.0, 1.0)
    glColor3f(0.0,1.0,0.0)                                      # 设置颜色为绿色
    glVertex3f( 1.0,-1.0, 1.0)
    glVertex3f(-1.0,-1.0, 1.0)
    glVertex3f(-1.0,-1.0,-1.0)
    glVertex3f( 1.0,-1.0,-1.0)
    glColor3f(0.0,0.0,1.0)                                      # 设置颜色为蓝色
    glVertex3f( 1.0, 1.0, 1.0)
    glVertex3f(-1.0, 1.0, 1.0)
    glVertex3f(-1.0,-1.0, 1.0)
    glVertex3f( 1.0,-1.0, 1.0)
    glColor3f(1.0,0.0,0.0)
    glVertex3f( 1.0,-1.0,-1.0)
    glVertex3f(-1.0,-1.0,-1.0)
    glVertex3f(-1.0, 1.0,-1.0)
    glVertex3f( 1.0, 1.0,-1.0)
    glColor3f(0.0,1.0,0.0)
    glVertex3f(-1.0, 1.0, 1.0)
    glVertex3f(-1.0, 1.0,-1.0)
    glVertex3f(-1.0,-1.0,-1.0)
    glVertex3f(-1.0,-1.0, 1.0)
    glColor3f(0.0,0.0,1.0)
    glVertex3f( 1.0, 1.0,-1.0)
    glVertex3f( 1.0, 1.0, 1.0)
    glVertex3f( 1.0,-1.0, 1.0)
    glVertex3f( 1.0,-1.0,-1.0)
    glEnd()
    glutSwapBuffers()                                           # 交换缓存
def InitGL(self, width, height):                                # OpenGL 初始化函数
    glClearColor(0.0, 0.0, 0.0, 0.0)                            # 设为黑色背景
    glClearDepth(1.0)                                           # 设置深度缓存
    glDepthFunc(GL_LESS)                                        # 设置深度测试类型
    glEnable(GL_DEPTH_TEST)                                     # 允许深度测试
    glShadeModel(GL_SMOOTH)                                     # 启动平滑阴影
```

```
        glMatrixMode(GL_PROJECTION)                                    # 设置观察矩阵
        glLoadIdentity()                                               # 重置观察矩阵
        gluPerspective(45.0, float(width)/float(height), 0.1, 100.0) # 计算屏幕高宽比
        glMatrixMode(GL_MODELVIEW)                                     # 设置观察矩阵
    def MainLoop(self):                                                # 进入消息循环
        glutMainLoop()
window = OpenGLWindow()                                                # 创建窗口
window.MainLoop()                                                      # 进入消息循环
```

运行 pyOpenGLDraw3D.py 脚本后，将创建如图 8-8 所示的窗口，并在窗口右侧绘制一个立方体，立方体的三个可视面具有不同的颜色。

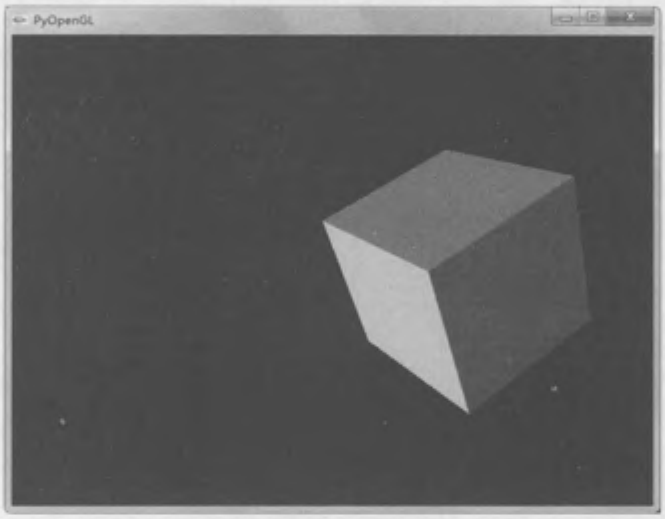


图 8-8　绘制三维图形

8.1.6　纹理映射

在 PyOpenGL 中进行纹理贴图需要使用 PIL 模块，在第 9 章中将讲解 PIL 模块的详细使用方法。下面所示的 pyOpenGLTexture.py 绘制了一个立方体，并对每一个面进行贴图。在 pyOpenGLTexture.py 脚本中使用 glutIdleFunc 函数设置了空闲时的场景绘制函数，创建了立方体旋转的动画。

```
# -*- coding:utf-8 -*-
# file: pyOpenGLTexture.py
#
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import sys
import Image
class OpenGLWindow:
    def __init__(self, width = 640, height = 480, title = 'PyOpenGL'):# 初始化
        glutInit(sys.argv)                                             # 传递命令行参数
        glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH) # 设置显示模式
```

```
        glutInitWindowSize(width, height)                          # 设置窗口大小
        self.window = glutCreateWindow(title)                      # 创建窗口
        glutDisplayFunc(self.Draw)                                 # 设置场景绘制函数
        glutIdleFunc(self.Draw)                                    # 设置空闲时场景绘制函数
        self.InitGL(width, height)                                 # 调用 OpenGL 初始化函数
        self.x = 0.2                                               # 旋转角度增量
        self.y = 0.2
        self.z = 0.2
    def Draw(self):                                                # 绘制场景
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)         # 清除屏幕
        glLoadIdentity()                                           # 重置观察矩阵
        glTranslatef(0.0,0.0,-5.0)                                 # 移动位置
        glRotatef(self.x,1.0,0.0,0.0)                              # 绕 X 轴旋转
        glRotatef(self.y,0.0,1.0,0.0)                              # 绕 Y 轴旋转
        glRotatef(self.z,0.0,0.0,1.0)                              # 绕 Z 轴旋转
        glBegin(GL_QUADS)                                          # 绘制立方体
        glTexCoord2f(0.0, 0.0)                                     # 对前面进行贴图
        glVertex3f(-1.0, -1.0,  1.0)
        glTexCoord2f(1.0, 0.0)
        glVertex3f( 1.0, -1.0,  1.0)
        glTexCoord2f(1.0, 1.0)
        glVertex3f( 1.0,  1.0,  1.0)
        glTexCoord2f(0.0, 1.0)
        glVertex3f(-1.0,  1.0,  1.0)
        glTexCoord2f(1.0, 0.0)                                     # 对后面进行贴图
        glVertex3f(-1.0, -1.0, -1.0)
        glTexCoord2f(1.0, 1.0)
        glVertex3f(-1.0,  1.0, -1.0)
        glTexCoord2f(0.0, 1.0)
        glVertex3f( 1.0,  1.0, -1.0)
        glTexCoord2f(0.0, 0.0)
        glVertex3f( 1.0, -1.0, -1.0)
        glTexCoord2f(0.0, 1.0)                                     # 对顶面进行贴图
        glVertex3f(-1.0,  1.0, -1.0)
        glTexCoord2f(0.0, 0.0)
        glVertex3f(-1.0,  1.0,  1.0)
        glTexCoord2f(1.0, 0.0)
        glVertex3f( 1.0,  1.0,  1.0)
        glTexCoord2f(1.0, 1.0)
        glVertex3f( 1.0,  1.0, -1.0)
        glTexCoord2f(1.0, 1.0)                                     # 对底面进行贴图
        glVertex3f(-1.0, -1.0, -1.0)
        glTexCoord2f(0.0, 1.0)
        glVertex3f( 1.0, -1.0, -1.0)
        glTexCoord2f(0.0, 0.0)
        glVertex3f( 1.0, -1.0,  1.0)
        glTexCoord2f(1.0, 0.0)
        glVertex3f(-1.0, -1.0,  1.0)
        glTexCoord2f(1.0, 0.0)                                     # 对右侧面进行贴图
        glVertex3f( 1.0, -1.0, -1.0)
        glTexCoord2f(1.0, 1.0)
        glVertex3f( 1.0,  1.0, -1.0)
        glTexCoord2f(0.0, 1.0)
        glVertex3f( 1.0,  1.0,  1.0)
```

```
        glTexCoord2f(0.0, 0.0)
        glVertex3f( 1.0, -1.0,  1.0)
        glTexCoord2f(0.0, 0.0)                                  # 对左侧面进行贴图
        glVertex3f(-1.0, -1.0, -1.0)
        glTexCoord2f(1.0, 0.0)
        glVertex3f(-1.0, -1.0,  1.0)
        glTexCoord2f(1.0, 1.0)
        glVertex3f(-1.0,  1.0,  1.0)
        glTexCoord2f(0.0, 1.0)
        glVertex3f(-1.0,  1.0, -1.0)
        glEnd()
        glutSwapBuffers()                                       # 交换缓存
        self.x = self.x + 0.2                                   # 旋转角度增加
        self.y = self.y + 0.2
        self.z = self.z + 0.2
    def InitGL(self, width, height):                            # OpenGL 初始化函数
        self.LoadTextures()                                     # 载入纹理
        glEnable(GL_TEXTURE_2D)                                 # 允许纹理映射
        glClearColor(0.0, 0.0, 0.0, 0.0)                        # 设为黑色背景
        glClearDepth(1.0)                                       # 设置深度缓存
        glDepthFunc(GL_LESS)                                    # 设置深度测试类型
        glEnable(GL_DEPTH_TEST)                                 # 允许深度测试
        glShadeModel(GL_SMOOTH)                                 # 启动平滑阴影
        glMatrixMode(GL_PROJECTION)                             # 设置观察矩阵
        glLoadIdentity()                                        # 重置观察矩阵
        gluPerspective(45.0, float(width)/float(height), 0.1, 100.0) # 计算屏幕高宽比
        glMatrixMode(GL_MODELVIEW)                              # 设置观察矩阵
    def LoadTextures(self):                                     # 载入纹理图片
        image = Image.open('python.bmp')                        # 打开图片
        width = image.size[0]                                   # 图像宽度
        height = image.size[1]                                  # 图像高度
        image = image.tostring('raw', 'RGBX', 0, -1)            # 转换图像
        glBindTexture(GL_TEXTURE_2D, glGenTextures(1))          # 创建纹理
        glPixelStorei(GL_UNPACK_ALIGNMENT,1)
        glTexImage2D(GL_TEXTURE_2D, 0, 3, width, height,
                0, GL_RGBA, GL_UNSIGNED_BYTE, image)
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP)
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST)
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST)
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL)
    def MainLoop(self):                                         # 进入消息循环
        glutMainLoop()
window = OpenGLWindow()                                         # 创建窗口
window.MainLoop()                                               # 进入消息循环
```

运行 pyOpenGLTexture.py 脚本后，将创建如图 8-9 所示的窗口，其中显示了一个立方体，在立方体的各面进行了贴图，并且该立方体还在不停地转动。



图 8-9 纹理贴图



必须要安装 PIL 模块后才能运行 pyOpenGLTexture.py 脚本，其安装方法将在第 9 章中介绍，为了运行这个脚本，可先参照第 9 章的内容进行安装。

8.2 播放音频文件

在 Windows 下，可以使用 PythonWin 提供的 win32com 模块来调用 DirectSound 播放和处理音频文件。除此之外，还可以直接调用 WMPlayer.OCX 来播放音频文件。

8.2.1 使用 DirectSound

DirectSound 是 DirectX API 的音频（waveaudio）组件之一，其提供了快速地混音及硬件加速功能，并且可以直接访问相关设备。DirectSound 可以进行波形声音的捕获、重放，也可以通过控制硬件和相应的驱动来获得更多的服务。

在 Python 中，可以通过 PythonWin 中的 win32com 模块来使用 DirectSound。由于使用 DirectSound 需要处理大量的数据，使得 Python 在运行速度上较慢，因此，此处仅给出一个简单地使用 DirectSound 播放 WAV 文件的例子。

下面所示的 pyDirectSound.py 是使用 DirectSound 播放 WAV 文件。

```
# -*- coding:utf-8 -*-
# file: pyDirectSound.py
#
import pywintypes                                          # 导入模块
import struct
from win32com.directsound import directsound
import tkinter
import tkinter.filedialog
WAV_HEADER_SIZE = struct.calcsize('<4sl4s4slhhllhh4sl')# 设置 WAV 头
class Window:
    def __init__(self):
        self.root = root = tkinter.Tk()                    # 创建组件
        buttonAdd = tkinter.Button(root, text = 'Add',
                command = self.add)
        buttonAdd.pack(side = 'left')
        buttonPlay = tkinter.Button(root, text = 'Play',
```

```
                command = self.play)
        buttonPlay.pack(side = 'left')
        buttonStop = tkinter.Button(root, text = 'Stop',
                command = self.stop)
        buttonStop.pack(side = 'left')
    def MainLoop(self):                                    # 进入消息循环
        self.root.mainloop()
    def add(self):                                         # 添加播放文件
        self.file = tkinter.filedialog.askopenfilename(
                title = 'Python DirectSound',
                filetypes=[('WAV', '*.wav')])
    def play(self):                                        # 播放文件
        print(self.file)
        f = open(self.file, 'rb')                          # 打开文件
        header = f.read(WAV_HEADER_SIZE)                   # 读取 WAV 文件头
        (riff, riffsize, wave, fmt, fmtsize,
            format, nchannels, samplespersecond,
            datarate, blockalign, bitspersample,
            data, size) =\
        struct.unpack('<4sl4s4slhhllhh4sl', header)        # 获取参数值
        print(riff, riffsize, wave, fmt, fmtsize,
            format, nchannels, samplespersecond,
            datarate, blockalign, bitspersample,
            data, size)
        if riff != 'RIFF' or fmtsize != 16 or fmt != 'fmt ' or data != 'data':
                                                           # 判断文件格式
            raise  Exception('Data Error')
        wfx = pywintypes.WAVEFORMATEX()                    # 创建 WAVEFORMATEX 结构
        wfx.wFormatTag = format
        wfx.nChannels = nchannels
        wfx.nSamplesPerSec = samplespersecond
        wfx.nAvgBytesPerSec = datarate
        wfx.nBlockAlign = blockalign
        wfx.wBitsPerSample = bitspersample
        d = directsound.DirectSoundCreate(None, None)  # 使用 DirectSound 播放声音
        d.SetCooperativeLevel(None, directsound.DSSCL_PRIORITY)
        sdesc = directsound.DSBUFFERDESC()
        sdesc.dwFlags = (
                directsound.DSBCAPS_STICKYFOCUS |
                directsound.DSBCAPS_CTRLPOSITIONNOTIFY
                )
        sdesc.dwBufferBytes = size
        sdesc.lpwfxFormat = wfx
        self.buffer = buffer = d.CreateSoundBuffer(sdesc, None)
        buffer.Update(0, f.read(size))
        buffer.Play(0)
    def stop(self):
        self.buffer.Stop()                                 # 停止
window = Window()
window.MainLoop()
```

8.2.2 使用 WMPlayer.OCX

使用 DirectSound 播放音频文件比较麻烦，而且需要自己对文件进行解码。由于 PythonWin 提供了对 COM 组件的支持，因此可以在 Python 中直接使用 WMPlayer.OCX 组件来播放音频文件。

下面所示的 pyMusicPlayer.py 使用 WMPlayer.OCX 组件创建了一个简单的音乐播放器。

```
# -*- coding:utf-8 -*-
# file: pyMusicPlayer.py
#
import tkinter                                          # 导入 tkinter 模块
import tkinter.filedialog                               # 导入 tkFileDialog 模块
from win32com.client import Dispatch
class Window:
    def __init__(self):
        self.root = root = tkinter.Tk()                 # 创建窗口
        buttonAdd = tkinter.Button(root, text = 'Add',
                command = self.add)
        buttonAdd.place(x = 150, y = 15)
        buttonPlay = tkinter.Button(root, text = 'Play',
                command = self.play)
        buttonPlay.place(x = 200, y = 15)
        buttonPause = tkinter.Button(root, text = 'Pause',
                command = self.pause)
        buttonPause.place(x= 250, y = 15)
        buttonStop = tkinter.Button(root, text = 'Stop',
                command = self.stop)
        buttonStop.place(x= 300, y = 15)
        buttonNext = tkinter.Button(root, text = 'Next',
                command = self.next)
        buttonNext.place(x = 350, y = 15)
        frame = tkinter.Frame(root, bd=2)
        self.playList = tkinter.Text(frame)
        scrollbar = tkinter.Scrollbar(frame)
        scrollbar.config(command=self.playList.yview)
        self.playList.pack(side = tkinter.LEFT)
        scrollbar.pack(side=tkinter.RIGHT, fill=tkinter.Y)
        frame.place(y = 50)
        self.wmp = Dispatch('WMPlayer.OCX')             # 绑定 WMPlayer.OCX
    def MainLoop(self):                                 # 进入消息循环
        self.root.minsize(510,380)
        self.root.maxsize(510,380)
        self.root.mainloop()
    def add(self):                                      # 添加播放文件
        file = tkinter.filedialog.askopenfilename(title = 'Python Music Player',
            filetypes=[('MP3', '*.mp3'),
                ('WMA', '*.wma'), ('WAV', '*.wav')])
        if file:
            media = self.wmp.newMedia(file)
            self.wmp.currentPlaylist.appendItem(media)
            self.playList.insert(tkinter.END, file + '\n')
    def play(self):
        self.wmp.controls.play()                        # 播放文件
    def pause(self):
        self.wmp.controls.pause()                       # 暂停
    def next(self):
        self.wmp.controls.next()                        # 下一首
    def stop(self):
        self.wmp.controls.stop()                        # 停止
window = Window()
window.MainLoop()
```

运行 pyMusicPlayer.py 显示如图 8-10 所示窗口，单击【Add】按钮，可以向播放列表中添加文件，单击【Play】按钮，将播放列表中的音乐。



图 8-10　使用 WMPlayer.OCX 播放音乐

8.3 PyGame

PyGame 是用来编写游戏的 Python 模块。PyGame 是基于 SDL 的，SDL（Simple DirectMedia Layer）是一个跨平台的多媒体开发包，SDL 专门为游戏和多媒体设计。使用 PyGame 可以方便地使用 SDL 库创建游戏和多媒体程序。

8.3.1 安装 PyGame

PyGame 是一个跨平台的 Python 模块，PyGame 的官方网站提供了 Windows 下的安装程序，下面以 Python 3.2 为例，介绍 PyGame 的安装过程，具体操作步骤如下。

step 1 从 PyGame 官方网站 http://www.pygame.org/下载 Windows 下的安装程序 pygame-1.9.2a0.win32-py3.2.msi。

step 2 双击运行安装程序，显示如图 8-11 所示界面。

step 3 单击【下一步】按钮，进入安装路径选择界面，第一项是以注册表中的信息确定 Python 3.2 的安装位置，如图 8-12 所示。



图 8-11　PyGame 安装程序

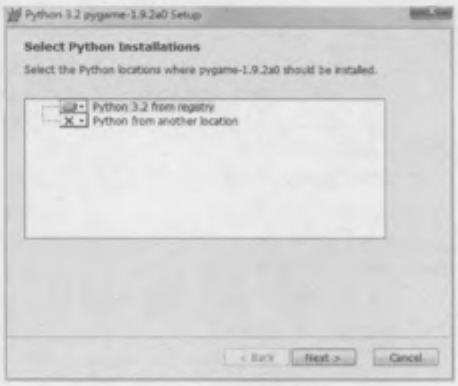


图 8-12　路径选择

step 4 单击【下一步】按钮开始安装，经过一段时间完成安装，将显示如图 8-13 所示界面，单击【Finish】按钮，完成 PyGame 的安装。



图 8-13 完成安装

在 PyGame 中，将对游戏各种功能的支持分成更小的模块，常用的模块如下。

- Display 提供了屏幕显示相关的函数。
- Event 提供了处理事件的函数。
- Image 提供了处理图片的函数。
- Key 提供了处理键盘按键的相关函数。
- Mouse 提供了处理鼠标消息的相关函数。
- Movie 提供了播放视频文件的相关函数，需要 PyMedia 的支持。
- Music 提供了播放音频文件的相关函数。
- Surface 提供了绘制屏幕的相关函数。
- Time 提供了处理时间的相关函数。

8.3.2 使用 PyGame 编写简单的游戏

使用 PyGame 模块可以很方便地绘制图片、播放声音、处理鼠标消息和键盘消息。通过 PyGame 可以很快地编写出简单的小游戏。下面所示的 pyGame.py 脚本编写了一个简单的石头、剪子、布的游戏。

```
# -*- coding:utf-8 -*-
# file: pyGame.py
#
import sys
import pygame
import threading
import random
class Game:                                                    # 创建游戏类
    def __init__(self):
        pygame.init()                                          # pygame 初始化
        self.screen = pygame.display.set_mode((800,600))       # 设置显示模式
        pygame.display.set_caption('Python Game')              # 设置窗口标题
```

```
        self.image = []                                          # 图片列表
        self.imagerect = []                                      # 图片大小列表
        self.vs = pygame.image.load('image/vs.gif')              # 载入图片
        self.o = pygame.image.load('image/o.gif')
        self.p = pygame.image.load('image/p.gif')
        self.u = pygame.image.load('image/u.gif')
        self.title = pygame.image.load('image/title.gif')
        self.start = pygame.image.load('image/start.gif')
        self.exit = pygame.image.load('image/exit.gif')
        for i in range(3):
            gif = pygame.image.load('image/' + str(i) + '.gif')
            self.image.append(gif)
        for i in range(3):                                       # 处理图片绘制区域
            image = self.image[i]
            rect = image.get_rect()
            rect.left = 200 * (i+1) + rect.left
            rect.top = 400
            self.imagerect.append(rect)
    def Start(self):                                             # 绘制游戏初始界面
        self.screen.blit(self.title, (200,100,400,140))          # 绘制游戏名称
        self.screen.blit(self.start, (350,300,100,50))           # 绘制开始按钮
        self.screen.blit(self.exit, (350, 400,100,50))           # 绘制退出按钮
        pygame.display.flip()                                    # 刷新屏幕
        start = 1
        while start:                                             # 进入消息循环
            for event in pygame.event.get():                     # 处理消息
                if event.type == pygame.QUIT:
                    sys.exit()
                elif event.type == pygame.MOUSEBUTTONDOWN:       # 处理鼠标单击消息
                    if self.isStart() == 0:
                        start = 0
                    elif self.isStart() == 1:
                        sys.exit()
                    else:
                        pass
                else:
                    pass
        self.run()                                               # 开始游戏
    def run(self):
        self.screen.fill((0,0,0))
        for i in range(3):                                       # 绘制图片
            self.screen.blit(self.image[i], self.imagerect[i])
        pygame.display.flip()                                    # 刷新屏幕进入消息循环
        while True:
            for event in pygame.event.get():                     # 处理消息
                if event.type == pygame.QUIT:
                    sys.exit()
                elif event.type == pygame.MOUSEBUTTONDOWN:       # 处理鼠标单击消息
                    self.OnMouseButDown()
                else:
                    pass
    def isStart(self):                                           # 判断鼠标单击的按钮
        pos = pygame.mouse.get_pos()
        if pos[0] > 350 and pos[0] < 450:
```

```
            if pos[1] > 300 and pos[1] < 350:
                return 0
            elif pos[1] > 400 and pos[1] < 450:
                return 1
            else:
                return 2
        else:
            return 2
    def OnMouseButDown(self):                                    # 处理鼠标单击消息
        self.screen.blit(self.vs, (300, 150, 140, 140))          # 绘制图片
        pos = pygame.mouse.get_pos()                             # 获取鼠标位置
        if pos[1] > 400 and pos[1] < 540:                        # 判断鼠标位置
            if pos[0] > 200 and pos[0] < 340:
                self.screen.blit(self.image[0],
                        (150 ,150, 140,140))
                self.isWin(0)
            elif pos[0] > 400 and pos[0] < 540:
                self.screen.blit(self.image[1],
                        (150 ,150, 140,140))
                self.isWin(1)
            elif pos[0] > 600 and pos[0] < 740:
                self.screen.blit(self.image[2],
                        (150 ,150, 140,140))
                self.isWin(2)
            else:
                pass
    def isWin(self, value):                                      # 判断谁赢
        num = random.randint(0, 2)                               # 产生随机数
        self.screen.blit(self.image[num],                        # 绘制相应图片
                (450 ,150, 590,240))
        pygame.display.flip()                                    # 刷新屏幕
        if num == value:                                         # 判断谁赢
            self.screen.blit(self.o,
                    (220, 10, 140, 70))
            pygame.display.flip()
        elif num < value:
            if num == 0:
                if value == 2:
                    self.screen.blit(self.u,
                            (220, 10, 140, 70))
                else:
                    self.screen.blit(self.p,
                            (220, 10, 140, 70))
                pygame.display.flip()
            else:
                self.screen.blit(self.u,
                        (220, 10, 140, 70))
                pygame.display.flip()
        else:
            if num == 2:
                if value == 1:
                    self.screen.blit(self.u,
                            (220, 10, 140, 70))
                else:
                    self.screen.blit(self.p,
                            (220, 10, 140, 70))
                pygame.display.flip()
```

```
            else:
                self.screen.blit(self.u,
                        (220, 10, 140, 70))
                pygame.display.flip()
game = Game()
game.Start()
```

运行 pyGame.py 脚本后，首先将显示如图 8-14 所示的游戏初始界面，单击【开始】按钮，显示如图 8-15 所示的游戏界面，下方显示了石头、剪刀、布这 3 种图形，可分别单击相应的图形出拳。例如，单击第 1 个图形（“石头”），脚本（计算机）将随机出一个图形，这时脚本将根据游戏者和计算机所出的图形进行判断胜负，并显示出结果，如图 8-16 所示。



图 8-14　游戏初始界面



图 8-15　游戏界面



图 8-16　显示胜负结果

8.4 本章小结

本章主要介绍了 Python 在多媒体编程方面的运用，包括三方面的主题：使用 PyOpenGL 绘制图形、播放音频文件和使用 PyGame 编写游戏。本章首先介绍了 PyOpenGL 模块的安装，接着介绍了使用 PyOpenGL 创建窗口使用、绘制文字、二维图形、三维图形，然后介绍了使用 DirectSound 和 WMPlayer 播放音频文件的方法，以及 PyGame 模块的使用，通过 PyGame 模块可快速开发出游戏程序。本章最后给出了一个简单游戏的示例，在本书第 26 章介绍了如何使用 PyGame 模块编写一个完整的游戏过程。

在下一章中将介绍 Python 的另一主题，即使用 PIL 处理图片。

◆　　◆　　◆

第 9 章　使用 PIL 处理图片

本章包括

- PIL 简介
- 使用 PIL 转换图片格式
- 使用 PIL 为图片添加 Logo
- 安装 PIL
- 使用 PIL 生成缩略图

Python Imaging Library（简称 PIL）为 Python 解释器提供了图像处理的功能。它提供了广泛的文件格式支持、高效的内部表示以及相当强大的图像处理功能。这 PIL 的核心被设计成为能够快速访问几种基本像素类型表示的图像数据。它为通用图像处理工具提供了一个坚实基础。通过使用 PIL 模块，使得 Python 可以对图片进行处理。例如，可用来对图片进行改变尺寸、格式、色彩、旋转等处理。

9.1　PIL 概述

由于 PIL 不是 Python 自带的模块，因此需要用户自己安装。PIL 是跨平台的，在 Windows 下可以使用 PIL 的强大功能。在 Windows 下安装 PIL 十分简便。

9.1.1　安装 PIL

1. 下载安装官方 PIL

可以从网站 http://www.pythonware.com/products/pil/下载 Windows 平台的 PIL 安装程序，目前还不支持 Python3，没有针对 Python 3 的安装程序，因此，下面以 Python2.5 为例，介绍其安装过程，具体步骤如下。

step 1　从 PIL 网站下载 Windows 平台下的安装程序 PIL-1.1.7.win32-py2.5.exe。

step 2　双击安装程序，显示如图 9-1 所示的安装向导。

step 3　单击【下一步】按钮，进入安装路径选择界面，如图 9-2 所示。



图 9-1　安装向导



图 9-2　选择路径

step 4 单击【下一步】按钮，进入安装确认界面，如图 9-3 所示。单击【下一步】按钮，完成 PIL 的安装。



图 9-3 确认安装

2. 下载安装非官方 PIL

虽然 PIL 官方目前还不支持 Python 3 的安装程序，但在互联网中还是可以找到一些支持 Python 3 的安装程序包。例如，在网站 http://www.lfd.uci.edu/~gohlke/pythonlibs/#pil 中就提供了针对 Python 3.2 的安装程序（这个网站还提供了一些其他模块的非官方版本），只是其模块名称为 Pillow-2.2.1.win32-py3.2.exe。虽然名称不是 PIL，但实际上是对 PIL 源码进行编译的。下载安装程序以后，就可以按以下步骤进行安装。

step 1 双击安装文件，显示如图 9-4 所示的安装向导。

step 2 单击【下一步】按钮，显示如图 9-5 所示界面，选择安装路径。



图 9-4 安装向导



图 9-5 选择路径

step 3 单击【下一步】按钮，显示如图 9-6 所示的确认安装界面，直接单击【下一步】按钮开始安装，经过一段时间，安装完成。



图 9-6 确认安装



在下载非官方安装程序时，网站 http://www.lfd.uci.edu/~gohlke/pythonlibs/#pil 中有一段提示文字，内容如下：

```
use `from PIL import Image` instead of `import Image`.
```

意思是说，使用下面的语句

```
from PIL import Image
```

代替原来的语句

```
import Image
```

9.1.2 PIL 简介

在 PIL 中，主要提供了以下对图片进行处理的模块。

- Image　　PIL 的主要模块。
- ImageChops　　图片计算模块。
- ImageColor　　颜色模块。
- ImageDraw　　绘图模块。
- ImageEnhance　　图片效果模块。
- ImageFile　　图片文件存取模块。
- ImageFileIO　　图片流模块。
- ImageFilter　　图片过滤模块。
- ImageFont　　字形模块，用于绘图。
- ImageGrab　　图片抓取模块。
- ImageOps　　图片处理模块。
- ImagePath　　路径队列模块。
- ImagePalette　　图片调色板模块。
- ImageSequence　　队列包装模块。
- ImageStat　　图片属性模块。
- ImageTk　　提供对 Tkinter 的支持。
- ImageWin　　提供对 Windows 的支持。
- PSDraw　　提供对 Postscript 的支持。

对于简单的图片处理，一般仅需使用 Image 模块，因此，这里仅对 Image 模块中的主要函数进行简要的介绍，其他模块中的函数可以参考 PIL 的帮助文档。

1. 打开图片

Image 模块中主要的函数是 open 函数，主要用于打开图片，其函数原型如下。

```
open(file, mode)
```

其参数含义如下。

- file　　要打开的图片文件。
- mode　　可选参数，打开文件的方式。

2. 复制图片

open 函数执行成功后返回一个 Image 对象，使用 Image 对象的 copy 方法可以复制图片，使用 crop 方法可以复制图片中的某一区域，其原型如下。

```
crop(box)
```

其参数含义如下。

- box　　一个由四个元素组成的元组，分别表示图片的左、上、右、下的位置。

3. 粘贴图片

使用 Image 对象的 paste 方法可以向图片中粘贴图片图像，paste 方法有以下几种形式。

```
paste(image, box)
paste(colour, box)
paste(image, box, mask)
paste(colour, box, mask)
```

其参数含义如下。

- image　　被粘贴到图片中的图片对象。
- box　　所要粘贴的区域，同 crop 中的参数 box。
- colour　　填充的颜色。
- mask　　指定填充颜色透明度。

4. 调整图片大小

使用 Image 对象的 resize 方法可以重新调整图片的大小。其原型如下。

```
resize(size, filter)
```

其参数含义如下。

- size　　图片调整后的大小，为由宽和高组成的元组。
- filter　　可选参数，可以是 NEAREST、BILINEAR、BICUBIC 或者 ANTIALIAS。

5. 旋转图片

使用 Image 对象的 rotate 方法可以旋转图片。其原型如下。

```
rotate(angle, filter, expand)
```

其参数含义如下。

- angle　　旋转的角度。
- filter　　可选参数，同 resize 中的 filter 参数。
- expand　　可选参数，如果为真则增大图片，容纳旋转后的图片，否则保持图片尺寸。

6. 显示图片

使用 Image 对象的 show 方法可以显示图片，在 Windows 下，Image 模块将调用 Windows 图片和传真查看器显示图片。使用 Image 对象的 save 方法可以保存图像。其原型如下。

```
save(outfile, format, options)
```

其参数含义如下。

- outfile　所保存的文件名。
- format　可选参数，保存的文件格式。若不给出，将根据保存的文件名的扩展名进行存储。
- options　其他的操作参数。

另外，Image 对象还有 size 属性，其为由宽和高组成的元组。Image 对象的 format 属性为图片的格式。Image 对象的 mode 属性为图片的模式。

9.2 使用 PIL 处理图片

PIL 提供了非常实用的函数，很多情况下仅需使用一两个 PIL 函数或者方法，就可以完成对图片的处理，如调整图片大小，转换图片格式等。配合 Python 的灵活性，使用 PIL 可以创建一些非常实用的图片处理脚本。

9.2.1 转换图片格式

使用 PIL 转换图片格式，主要使用 PIL 的 Image 模块。首先使用 Image.open 函数打开文件，然后将文件保存为所需要的格式即可。Image 可以根据文件的扩展名自动选择文件保存的格式，因此不需要设置文件格式。

下面所示的 pyImageConv.py 脚本使用 Image 模块进行批量图片文件格式转换。

```
# -*- coding:utf-8 -*-
# file: pyImageConv.py
#
import os                                              # 导入模块
from PIL import Image
import tkinter
import tkinter.filedialog
import tkinter.messagebox
class Window:                                          # 创建窗口
    def __init__(self):
        self.root = root = tkinter.Tk()                # 创建组件
        label = tkinter.Label(root, text = '选择目录')
        label.place(x = 5, y = 5)
        self.entry = tkinter.Entry(root)
        self.entry.place(x=60, y = 5)
        self.buttonBrowser = tkinter.Button(root, text = '浏览',
                command = self.Browser)
        self.buttonBrowser.place(x=210, y = 5)
        frameF = tkinter.Frame(root)
        frameF.place(x = 5, y = 30)
```

```
        frameT = tkinter.Frame(root)
        frameT.place(x = 100, y = 30)
        self.fImage = tkinter.StringVar()              # 生成关联变量
        self.tImage = tkinter.StringVar()
        self.status = tkinter.StringVar()
        self.fImage.set('.bmp')
        self.tImage.set('.bmp')
        labelFrom = tkinter.Label(frameF, text = 'From')
        labelFrom.pack(anchor='w')
        labelTo = tkinter.Label(frameT, text = 'To')
        labelTo.pack(anchor='w')
        frBmp = tkinter.Radiobutton(frameF, variable = self.fImage,
                value = '.bmp', text = 'BMP' )
        frBmp.pack(anchor='w')
        frJpg = tkinter.Radiobutton(frameF, variable = self.fImage,
                value = '.jpg', text = 'JPG' )
        frJpg.pack(anchor='w')
        frGif = tkinter.Radiobutton(frameF, variable = self.fImage,
                value = '.gif', text = 'GIF' )
        frGif.pack(anchor='w')
        frPng = tkinter.Radiobutton(frameF, variable = self.fImage,
                value = '.png', text = 'PNG' )
        frPng.pack(anchor='w')
        trBmp = tkinter.Radiobutton(frameT, variable = self.tImage,
                value = '.bmp', text = 'BMP' )
        trBmp.pack(anchor='w')
        trJpg = tkinter.Radiobutton(frameT, variable = self.tImage,
                value = '.jpg', text = 'JPG' )
        trJpg.pack(anchor='w')
        trGif = tkinter.Radiobutton(frameT, variable = self.tImage,
                value = '.gif', text = 'GIF' )
        trGif.pack(anchor='w')
        trPng = tkinter.Radiobutton(frameT, variable = self.tImage,
                value = '.png', text = 'PNG' )
        trPng.pack(anchor='w')
        self.buttonConv = tkinter.Button(root, text = '转换',
                command = self.Conv)
        self.buttonConv.place(x=80, y = 160)
        self.labelStatus = tkinter.Label(root,textvariable=self.status)
        self.labelStatus.place(x=50, y = 195)
    def MainLoop(self):                                # 进入消息循环
        self.root.minsize(250,220)
        self.root.maxsize(250,220)
        self.root.mainloop()
    def Browser(self):                                 # 浏览目录
        directory = tkinter.filedialog.askdirectory(title='Python')
        if directory:
            self.entry.delete(0, tkinter.END)
            self.entry.insert(tkinter.END, directory)
    def Conv(self):                                    # 转换文件格式
        n = 0
        t = self.tImage.get()
        f = self.fImage.get()
        path = self.entry.get()
        if path == '':
            tkinter.messagebox.showerror('Python tkinter','请输入路径')
            return
        filenames = os.listdir(path)
```

```
        os.mkdir(path + '/' + t[-3:])
        for filename in filenames:
            if filename[-4:] == f:
                Image.open(path + '/' +filename).save(path +
                        '/' + t[-3:] + '/' + filename[:-4] + t)
                n = n + 1
        self.status.set('成功转换%d张图片' % n)
window = Window()
window.MainLoop()
```

运行 pyImageConv.py 脚本后，将显示如图 9-7 所示窗口，单击【浏览】按钮，选择一个保存源图片的目录，接着在下方选择从一种图片格式转换为另一种图片格式，最后单击【转换】按钮，即可批量转换图片的格式。转换操作完成后，在指定目录下新建一个目录，用来保存转换后的图片。



图 9-7 使用 PIL 转换文件格式

9.2.2 生成缩略图

使用 PIL 生成缩略图主要是使用 Image 的 resize 函数。使用 resize 函数可以重新指定图片的大小。下面所示的 pyImageThumb.py 脚本，就是使用 PIL 模块批量生成图片的缩略图。

```
# -*- coding:utf-8 -*-
# file: pyImageThumb.py
#
import os                                                    # 导入模块
from PIL import Image
import tkinter
import tkinter.filedialog
import tkinter.messagebox

class Window:
    def __init__(self):
        self.root = root = tkinter.Tk()                      # 创建窗口
        self.Image = tkinter.StringVar()
        self.status = tkinter.StringVar()
        self.mstatus = tkinter.IntVar()
        self.fstatus = tkinter.IntVar()
        self.Image.set('bmp')
        self.mstatus.set(0)
        self.fstatus.set(0)
        label = tkinter.Label(root, text = '输入百分比')
        label.place(x = 5, y = 5)
        self.entryNew = tkinter.Entry(root)
        self.entryNew.place(x = 70, y = 5)
        self.checkM = tkinter.Checkbutton(root, text ='批量转换',
                command = self.OnCheckM,
```

```
            variable = self.mstatus,
            onvalue = 1,
            offvalue = 0)
        self.checkM.place(x = 5, y = 30)
        label = tkinter.Label(root, text = '选择文件')
        label.place(x = 5, y = 55)
        self.entryFile = tkinter.Entry(root)
        self.entryFile.place(x=60, y = 55)
        self.buttonBrowserFile = tkinter.Button(root, text = '浏览',
            command = self.BrowserFile)
        self.buttonBrowserFile.place(x=200, y = 55)
        label = tkinter.Label(root, text = '选择目录')
        label.place(x = 5, y = 80)
        self.entryDir = tkinter.Entry(root,
            state = tkinter.DISABLED)
        self.entryDir.place(x=60, y = 80)
        self.buttonBrowserDir = tkinter.Button(root, text = '浏览',
            command = self.BrowserDir,
            state = tkinter.DISABLED)
        self.buttonBrowserDir.place(x=200, y = 80)

        self.checkF = tkinter.Checkbutton(root, text ='改变文件格式',
            command = self.OnCheckF,
            variable = self.fstatus,
            onvalue = 1,
            offvalue = 0)
        self.checkF.place(x = 5, y = 110)
        frame = tkinter.Frame(root)
        frame.place(x = 10, y = 130)
        labelTo = tkinter.Label(frame, text = 'To')
        labelTo.pack(anchor='w')
        self.rBmp = tkinter.Radiobutton(frame, variable = self.Image,
            value = 'bmp', text = 'BMP',
            state = tkinter.DISABLED)
        self.rBmp.pack(anchor='w')
        self.rJpg = tkinter.Radiobutton(frame, variable = self.Image,
            value = 'jpg', text = 'JPG',
            state = tkinter.DISABLED)
        self.rJpg.pack(anchor='w')
        self.rGif = tkinter.Radiobutton(frame, variable = self.Image,
            value = 'gif', text = 'GIF',
            state = tkinter.DISABLED)
        self.rGif.pack(anchor='w')
        self.rPng = tkinter.Radiobutton(frame, variable = self.Image,
            value = 'png', text = 'PNG',
            state = tkinter.DISABLED)
        self.rPng.pack(anchor='w')
        self.buttonConv = tkinter.Button(root, text = '转换',
            command = self.Conv)
        self.buttonConv.place(x=100, y = 175)
        self.labelStatus = tkinter.Label(root,textvariable=self.status)
        self.labelStatus.place(x=80, y = 205)
    def MainLoop(self):                                    # 进入消息循环
        self.root.minsize(250,270)
        self.root.maxsize(250,250)
        self.root.mainloop()
    def BrowserDir(self):                                  # 选择路径
```

```
        directory = tkinter.filedialog.askdirectory(title='Python')
        if directory:
            self.entryDir.delete(0, tkinter.END)
            self.entryDir.insert(tkinter.END, directory)
    def BrowserFile(self):                                  # 选择文件
        file = tkinter.filedialog.askopenfilename(title = 'Python Music Player',
            filetypes=[('JPG', '*.jpg'), ('BMP', '*.bmp'),
                ('GIF', '*.gif'), ('PNG', '*.png')])
        if file:
            self.entryFile.delete(0, tkinter.END)
            self.entryFile.insert(tkinter.END, file)
    def OnCheckM(self):                                     # 设置组件状态
        if not self.mstatus.get():
            self.entryDir.config(state = tkinter.DISABLED)
            self.entryFile.config(state = tkinter.NORMAL)
            self.buttonBrowserDir.config(state = tkinter.DISABLED)
            self.buttonBrowserFile.config(state = tkinter.NORMAL)
        else:
            self.entryDir.config(state = tkinter.NORMAL)
            self.entryFile.config(state = tkinter.DISABLED)
            self.buttonBrowserDir.config(state = tkinter.NORMAL)
            self.buttonBrowserFile.config(state = tkinter.DISABLED)
    def OnCheckF(self):                                     # 设置组件状态
        if not self.fstatus.get():
            self.rBmp.config(state = tkinter.DISABLED)
            self.rJpg.config(state = tkinter.DISABLED)
            self.rGif.config(state = tkinter.DISABLED)
            self.rPng.config(state = tkinter.DISABLED)
        else:
            self.rBmp.config(state = tkinter.NORMAL)
            self.rJpg.config(state = tkinter.NORMAL)
            self.rGif.config(state = tkinter.NORMAL)
            self.rPng.config(state = tkinter.NORMAL)
    def Conv(self):                                         # 转换图片
        n = 0
        if self.mstatus.get():
            path = self.entryDir.get()
            if path == '':
                tkinter.messagebox.showerror('Python tkinter','请输入路径')
                return
            filenames = os.listdir(path)
            if self.fstatus.get():
                f = self.Image.get()
                for filename in filenames:
                    if filename[-3:] in ('bmp','jpg','gif','png'):
                        self.make(path + '/' + filename, f)
                        n = n + 1
            else:
                for filename in filenames:
                    if filename[-3:] in ('bmp','jpg','gif','png'):
                        self.make(path + '/' + filename)
                        n = n + 1
        else:
            file = self.entryFile.get()
            if file == '':
                tkinter.messagebox.showerror('Python tkinter','请选择文件')
                return
            if self.fstatus.get():
```

```
                f = self.Image.get()
                self.make(file, f)
                n = n + 1
            else:
                self.make(file)
                n = n + 1
        self.status.set('成功转换%d 图片' % n)
    def make(self, file, format = None):                    # 生成缩略图
        im = Image.open(file)
        mode = im.mode
        if mode not in ('L', 'RGB'):
            im = im.convert('RGB')
        width, height = im.size
        s = self.entryNew.get()
        if s == '':
            tkMessageBox.showerror('Python tkinter','请输入百分比')
            return
        else:
            n = int(s)
        nwidth = int(width * n / 100)
        nheight = int(height * n / 100)
        thumb = im.resize((nwidth,nheight), Image.ANTIALIAS)
        if format:
            thumb.save(file[:(len(file) - 4)] + '_thumb.' + format)
        else:
            thumb.save(file[:(len(file) - 4)] + '_thumb' + file[-4:])
window = Window()
window.MainLoop()
```

运行 pyImageThumb.py 后，将显示如图 9-8 所示的窗口，输入缩略图的百分比，接着选择一个文件或目录进行转换，单击【转换】按钮，即可为选择的图片或选择目录中所有的图片生成缩略图。生成的缩略图将与原图保存在同一个目录中，缩略图的文件名由原文件名加上“_thumb.jpg”组成。



图 9-8　使用 PIL 生成缩略图

9.2.3 为图片添加 Logo

使用 PIL 为图片添加 Logo，主要是使用 Image 的 paste 函数。paste 函数可以向图片中粘贴其他图片。下面所示的 pyImageAddLogo.py 脚本是使用 PIL 模块为图片批量添加 Logo。

```
# -*- coding:utf-8 -*-
# file: pyImageAddLogo.py
#
```

```
import os                                                    # 导入模块
from PIL import Image
import tkinter
import tkinter.filedialog
import tkinter.messagebox

class Window:
    def __init__(self):
        self.root = root = tkinter.Tk()                      # 创建窗口
        self.Image = tkinter.StringVar()
        self.status = tkinter.StringVar()
        self.mstatus = tkinter.IntVar()
        self.fstatus = tkinter.IntVar()
        self.pstatus = tkinter.IntVar()
        self.Image.set('bmp')
        self.mstatus.set(0)
        self.fstatus.set(0)
        self.pstatus.set(0)
        label = tkinter.Label(root, text = 'Logo')
        label.place(x = 5, y = 5)
        self.entryLogo = tkinter.Entry(root)
        self.entryLogo.place(x = 50, y = 5)
        self.buttonBrowserLogo = tkinter.Button(root, text = '浏览',
                command = self.BrowserLogo)
        self.buttonBrowserLogo.place(x = 200, y = 5)
        self.checkM = tkinter.Checkbutton(root, text ='批量转换',
                command = self.OnCheckM,
                variable = self.mstatus,
                onvalue = 1,
                offvalue = 0)
        self.checkM.place(x = 5, y = 30)
        label = tkinter.Label(root, text = '选择文件')
        label.place(x = 5, y = 55)
        self.entryFile = tkinter.Entry(root)
        self.entryFile.place(x = 60, y = 55)
        self.buttonBrowserFile = tkinter.Button(root, text = '浏览',
                command = self.BrowserFile)
        self.buttonBrowserFile.place(x=200, y = 55)
        label = tkinter.Label(root, text = '选择目录')
        label.place(x = 5, y = 80)
        self.entryDir = tkinter.Entry(root,
                state = tkinter.DISABLED)
        self.entryDir.place(x=60, y = 80)
        self.buttonBrowserDir = tkinter.Button(root, text = '浏览',
                command = self.BrowserDir,
                state = tkinter.DISABLED)
        self.buttonBrowserDir.place(x=200, y = 80)

        self.checkF = tkinter.Checkbutton(root, text ='改变文件格式',
                command = self.OnCheckF,
                variable = self.fstatus,
                onvalue = 1,
                offvalue = 0)
        self.checkF.place(x = 5, y = 110)
        frame = tkinter.Frame(root)
        frame.place(x = 10, y = 130)
        labelTo = tkinter.Label(frame, text = '格式')
```

```
        labelTo.pack(anchor='w')
        self.rBmp = tkinter.Radiobutton(frame, variable = self.Image,
                value = 'bmp', text = 'BMP',
                state = tkinter.DISABLED)
        self.rBmp.pack(anchor='w')
        self.rJpg = tkinter.Radiobutton(frame, variable = self.Image,
                value = 'jpg', text = 'JPG',
                state = tkinter.DISABLED)
        self.rJpg.pack(anchor='w')
        self.rGif = tkinter.Radiobutton(frame, variable = self.Image,
                value = 'gif', text = 'GIF',
                state = tkinter.DISABLED)
        self.rGif.pack(anchor='w')
        self.rPng = tkinter.Radiobutton(frame, variable = self.Image,
                value = 'png', text = 'PNG',
                state = tkinter.DISABLED)
        self.rPng.pack(anchor='w')
        pframe = tkinter.Frame(root)
        pframe.place(x = 70, y = 130)
        labelPos = tkinter.Label(pframe, text = '位置')
        labelPos.pack(anchor = 'w')
        self.rLT = tkinter.Radiobutton(pframe, variable = self.pstatus,
                value = 0, text = '左上角')
        self.rLT.pack(anchor = 'w')
        self.rRT = tkinter.Radiobutton(pframe, variable = self.pstatus,
                value = 1, text = '右上角')
        self.rRT.pack(anchor = 'w')
        self.rLB = tkinter.Radiobutton(pframe, variable = self.pstatus,
                value = 2, text = '左下角')
        self.rLB.pack(anchor = 'w')
        self.rRB = tkinter.Radiobutton(pframe, variable = self.pstatus,
                value = 3, text = '右下角')
        self.rRB.pack(anchor = 'w')
        self.buttonAdd = tkinter.Button(root, text = '添加',
                command = self.Add)
        self.buttonAdd.place(x=180, y = 175)
        self.labelStatus = tkinter.Label(root,textvariable=self.status)
        self.labelStatus.place(x=150, y = 205)
    def MainLoop(self):                                    # 进入消息循环
        self.root.minsize(250,270)
        self.root.maxsize(250,270)
        self.root.mainloop()
    def BrowserLogo(self):
        file = tkinter.filedialog.askopenfilename(title = 'Python Music Player',
            filetypes=[('JPG', '*.jpg'), ('BMP', '*.bmp'),
                ('GIF', '*.gif'), ('PNG', '*.png')])
        if file:
            self.entryLogo.delete(0, tkinter.END)
            self.entryLogo.insert(tkinter.END, file)
    def BrowserDir(self):                                  # 选择路径
        directory = tkinter.filedialog.askdirectory(title='Python')
        if directory:
            self.entryDir.delete(0, tkinter.END)
            self.entryDir.insert(tkinter:END, directory)
    def BrowserFile(self):                                 # 选择文件
        file = tkinter.filedialog.askopenfilename(title = 'Python Music Player',
            filetypes=[('JPG', '*.jpg'), ('BMP', '*.bmp'),
```

```
            ('GIF', '*.gif'), ('PNG', '*.png')])
        if file:
            self.entryFile.delete(0, tkinter.END)
            self.entryFile.insert(tkinter.END, file)
    def OnCheckM(self):                                    # 设置组件状态
        if not self.mstatus.get():
            self.entryDir.config(state = tkinter.DISABLED)
            self.entryFile.config(state = tkinter.NORMAL)
            self.buttonBrowserDir.config(state = tkinter.DISABLED)
            self.buttonBrowserFile.config(state = tkinter.NORMAL)
        else:
            self.entryDir.config(state = tkinter.NORMAL)
            self.entryFile.config(state = tkinter.DISABLED)
            self.buttonBrowserDir.config(state = tkinter.NORMAL)
            self.buttonBrowserFile.config(state = tkinter.DISABLED)
    def OnCheckF(self):                                    # 设置组件状态
        if not self.fstatus.get():
            self.rBmp.config(state = tkinter.DISABLED)
            self.rJpg.config(state = tkinter.DISABLED)
            self.rGif.config(state = tkinter.DISABLED)
            self.rPng.config(state = tkinter.DISABLED)
        else:
            self.rBmp.config(state = tkinter.NORMAL)
            self.rJpg.config(state = tkinter.NORMAL)
            self.rGif.config(state = tkinter.NORMAL)
            self.rPng.config(state = tkinter.NORMAL)
    def Add(self):                                         # 处理图片
        n = 0
        if self.mstatus.get():
            path = self.entryDir.get()
            if path == '':
                tkinter.messagebox.showerror('Python tkinter','请输入路径')
                return
            filenames = os.listdir(path)
            if self.fstatus.get():
                f = self.Image.get()
                for filename in filenames:
                    if filename[-3:] in ('bmp','jpg','gif','png'):
                        self.addlogo(path + '/' + filename, f)
                        n = n + 1
            else:
                for filename in filenames:
                    if filename[-3:] in ('bmp','jpg','gif','png'):
                        self.addlogo(path + '/' + filename)
                        n = n + 1
        else:
            file = self.entryFile.get()
            if file == '':
                tkinter.messagebox.showerror('Python tkinter','请选择文件')
                return
            if self.fstatus.get():
                f = self.Image.get()
                self.addlogo(file, f)
                n = n + 1
            else:
                self.addlogo(file)
                n = n + 1
        self.status.set('成功添加%d张图片' % n)
```

```
    def addlogo(self, file, format = None):                    # 向图片添加 Logo
        logo = self.entryLogo.get()
        if logo == '':
            tkinter.messagebox.showerror('Python tkinter','请选择 Logo')
            return
        im = Image.open(file)
        lo = Image.open(logo)
        imwidth = im.size[0]
        imheight = im.size[1]
        lowidth = lo.size[0]
        loheight = lo.size[1]
        pos = self.pstatus.get()
        if pos == 0:
            left = 0
            top = 0
            right = lowidth
            bottom = loheight
        elif pos == 1:
            left = imwidth - lowidth
            top = 0
            right = imwidth
            bottom = loheight
        elif pos == 2:
            left = 0
            top = imheight - loheight
            right = lowidth
            bottom = imheight
        else:
            left = imwidth - lowidth
            top = imheight - loheight
            right = imwidth
            bottom = imheight
        im.paste(lo, (left, top, right, bottom))
        if format:
            im.save(file[:(len(file) - 4)] + '_logo.' + format)
        else:
            im.save(file[:(len(file) - 4)] + '_logo' + file[-4:])
window = Window()
window.MainLoop()
```

运行 pyImageAddLogo.py 脚本后，将显示如图 9-9 所示的窗口，选择一个作为 Logo 的图片，接着选择需要添加 Logo 的图片，然后在窗口下方选择 Logo 添加的位置，最后单击【添加】按钮，即可将 Logo 图片添加至图片的指定位置。



图 9-9　使用 PIL 为图片添加 Logo

9.3 本章小结

本章介绍了一个 Python 处理图片的模块 PIL，PIL 模块的功能非常强大，可对图片进行多种操作。在本章中首先介绍了 PIL 模块的下载和安装。由于 PIL 目前没有针对 Python 3 的安装包，因此本章另外给出了一个非官方的下载包，安装后，即可在 Python 3 中使用 PIL 模块了。最后本章介绍了使用 PIL 处理图片的几个案例：使用 PIL 转换图片格式、生成缩略图、为图片添加 Logo。

在下一章中将介绍 Python 在系统编程方面的应用，还将介绍将 Python 脚本打包为 Exe 程序的方法。

◆ ◆ ◆

第10章 系统编程

本章包括

- 使用 Python 操作注册表
- 使用 py2exe 生成可执行文件
- 在 Python 中运行其他程序
- 用 Python 操作文件和目录
- 使用 cx_freeze 生成可执行文件

Python 虽然是脚本语言，但借助其扩展后同样可以进行系统级别的程序编写。在这一章中，将使用 PythonWin 提供的 Windows API 函数接口，编写与系统相关的 Python 脚本。使用 Python 脚本与使用 VC++编写的应用程序在功能上没有什么区别，而且使用 Python 还省去了编译、连接的过程。使用 Python 开发一些实用的脚本更为迅速，在代码上更加简洁。

10.1 访问 Windows 注册表

通过使用 win32api 模块，Python 可以方便地访问注册表，并对其进行打开、关闭、添加项、删除项，以及进行添加、修改项值等操作。

10.1.1 注册表概述

Windows 注册表是 Windows 系统用于存储用户、应用程序和硬件设备配置系统所必需信息的数据库。Windows 注册表中包含了系统在运行期间需要查询的信息，如用户的配置文件、系统中的应用程序以及计算机的硬件信息等。Windows 注册表中的信息影响着系统的运行，因此注册表也成了 Windows 系统中的“是非之地”。Windows 注册表由 5 个基本项组成，如表 10-1 所示。

表 10-1 Windows 注册表基本项

项名	描述
HKEY_CLASSES_ROOT	是 HKEY_LOCAL_MACHINE\Software 的子项，保存打开文件所对应的应用程序信息
HKEY_CURRENT_USER	是 HKEY_USERS 的子项，保存当前用户的配置信息
HKEY_LOCAL_MACHINE	保存计算机的配置信息，针对所有用户
HKEY_USERS	保存计算机上的所有以活动方式加载的用户配置文件
HKEY_CURRENT_CONFIG	保存计算机的硬件配置文件信息

对 Windows 注册表进行操作的基本的步骤是：首先导入 win32api 和 win32con 模块，打开要进行操作的项，获得该项的句柄，然后执行相关的操作，如读、写、修改等，完成操作后，还需要关闭注册表，以释放资源，具体流程如图 10-1 所示。



由于Windows注册表中存放着系统中的重要数据，因此在使用以下代码对注册表进行操作前应备份注册表，以免误操作导致系统崩溃。

下面是一个操作注册表的简单例子。

```
>>> import win32api   #导入模块
>>> import win32con
# 项的具体值由安装的 Python 版本决定
>>> name = 'SOFTWARE\\Python\\PythonCore\\2.5\\InstallPath'
# 打开要注册表，获得要进行操作的项的句柄
>>> key =
win32api.RegOpenKey(win32con.HKEY_LOCAL_MACHINE,name,0,win32con.KEY_ALL_ACCESS
)
>>> win32api.RegQueryValue(key,'') # 进行操作，这里读取项的默认值
'C:\\Python32\\'                   # 输出项的默认值
>>> win32api.RegCloseKey(key)      # 关闭注册表，结束操作
```

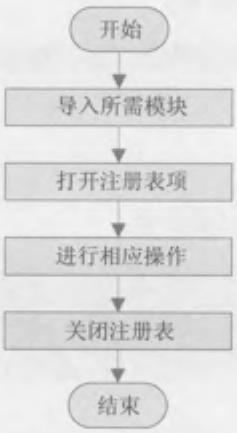


图 10-1 注册表操作基本流程

10.1.2 使用 Python 操作注册表

注册表操作的相关函数可以分为打开注册表、关闭注册表、读取项值、设置添加项值、添加项，以及删除项等几类。

1. 打开注册表

对注册表进行操作前，必须打开注册表。在 Python 中，可以使用以下两个函数：RegOpenKey 和 RegOpenKeyEx，其函数原型分别如下。

```
RegOpenKey(key, subKey , reserved , sam)
RegOpenKeyEx(key, subKey , reserved , sam)
```

两个函数的参数一样，参数含义如下。

- key　　必须为表 10-1 中列出的项。

- subKey　　要打开的子项。
- reserved　　必须为 0。
- sam　　对打开的子项进行的操作，包括 win32con.KEY_ALL_ACCESS、win32con.KEY_READ、win32con. KEY_WRITE 等。

以下代码可以打开注册表“HKEY_CURRENT_USER\Software”项。

```
>>> import win32api      #导入 win32api 模块
>>> import win32con      #导入 win32con 模块
# 使用 RegOpenKey 打开注册表项
>>> key =
win32api.RegOpenKey(win32con.HKEY_CURRENT_USER,'Software',0,win32con.KEY_READ)
>>> print(key)           # key 为打开的项的句柄
<PyHKEY:260>
```

2. 关闭注册表

对于打开的注册表，在操作完成后，需要进行关闭操作。在 Python 中，使用 RegCloseKey 函数可关闭打开的注册表项。其函数原型如下。

```
RegCloseKey(key)
```

其参数只有一个，其含义如下。

- key　已经打开的注册表项。
- 以下代码关闭一个已经打开的注册表项。

```
# 关闭刚才打开的注册表项
>>> win32api.RegCloseKey(key)
>>> print(key)
<PyHKEY:0>
```

3. 读取项值

在打开注册表项以后，可以使用 RegQueryValue 函数读取项的默认值。如果要读取某一项值，则可以使用 RegQueryValueEx 函数。其函数原型分别如下。

```
RegQueryValue(key, subKey )
RegQueryValueEx(key, valueName )
```

对于 RegQueryValue，其参数含义如下。

- key　　已打开的注册表项的句柄。
- subKey　　要操作的子项。

对于 RegQueryValueEx，其参数含义如下。

- key　　已经打开的注册表项的句柄。
- valueName　　要读取的项值名称。

以下代码可以实现对“HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer”项

的操作。

```
>>> import win32api
>>> import win32con
# 打开 "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer" 项
>>> key =
win32api.RegOpenKey(win32con.HKEY_LOCAL_MACHINE,'SOFTWARE\\Microsoft\\Internet
Explorer',0, win32con.KEY_ALL_ACCESS)
>>> win32api.RegQueryValue(key,'') # 读取项的默认值
''                                 # 输出为空，表示其默认值未设置
#读取项值名称为 Version 的项值数据，也就是 Internet Explorer 的版本
>>> win32api.RegQueryValueEx(key,'Version')
('9.10.9200.16736', 1)
>>> win32api.RegQueryInfoKey(key)  #  RegQueryInfoKey 函数查询项的基本信息
(32, 10, 130288102031220711)       #  返回项的子项数目、项值数目，以及最后一次修改时间
```

4. 设置项值

要修改或者重新设置注册表某一项的项值，可以使用 RegSetValueEx 函数，如果要设置项的默认值，则可以使用 RegSetValue。需要说明的是，对于 RegSetValueEx，如果要设置的项值不存在，那么 RegSetValueEx 会添加该项值；如果存在，则修改该项值。其函数原型分别如下。

```
RegSetValueEx(key, valueName, reserved, type, value)
RegSetValue(key, subKey, type, value)
```

对于 RegSetValueEx，其参数含义如下。

- key　　要设置的项的句柄。
- valuaName　　要设置的项值名称。
- reserved　　保留，可以设为 0。
- type　　项值的类型。
- value　　所要设置的值。

对于 RegSetValue 函数，其参数的含义如下。

- key　　已经打开的项的句柄。
- subKey　　所要设置的项。
- type　　项值的类型，必须为 win32con.REG_SZ。
- value　　项值数据，为字符串。

以下代码可以实现修改 "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer" 的默认值，以及其 "Version" 项值数据。

```
# 将 "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer" 的默认值设为 python
>>> win32api.RegSetValue(key,'',win32con.REG_SZ,'python')
# 将其 "Version" 设置为 7.0.2900.2180
>>> win32api.RegSetValueEx(key,'Version',0,win32con.REG_SZ,'7.0.2900.2180')
```

5. 添加、删除项

要向注册表中添加项，可以使用函数 RegCreateKey。而 RegDeleteKey 函数可以删除注册表

中的项。这两个函数的原型分别如下。

```
RegCreateKey(key, subKey )
RegDeleteKey (key, subKey )
```

其参数含义相同，参数含义分别如下。

- key　　已经打开的注册表项的句柄。
- subKey　　所要操作（添加或删除）的子项。

以下的代码可以实现对“HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer”项的添加、删除子项操作。

```
# 向 "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer" 添加子项 "Python"
>>> win32api.RegCreateKey(key,'Python')
<PyHKEY:336>  # 新创建的子项的句柄
# 删除刚才创建的子项 "Python"
>>> win32api.RegDeleteKey(key,'Python')
```

10.1.3 查看系统启动项

很多流氓软件、木马，以及病毒等都会随着系统的启动而运行，它们通常是采取修改注册表的方法，将自身路径添加到注册表中，以达到开机运行的目的。

下面的 Python 脚本通过查询注册表中以下几项的值，可以列举出随系统启动的可执行文件，以便于查找可疑项目，从而清除系统中的恶意软件。

- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run。
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce。
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnceEx。
- HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run。
- HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce。

```
#  -*- coding: utf-8 -*-
#  file: AutoRuns.py
#
# 导入所需要的模块
from win32api import *
from win32con import *
# GetValues 函数用于获得某注册表项下所有的项值
def GetValues(fullname):
    # 把完整的项拆分成根项和子项两部分
    name = str.split(fullname,'\\',1)
    #当注册表中没有某项时将抛出异常
    try:
        # 打开相应的项，为了让该函数更通用
        # 使用了多个判断语句
        if name[0] == 'HKEY_LOCAL_MACHINE':
            key = RegOpenKey(HKEY_LOCAL_MACHINE,name[1],0,KEY_READ)
        elif name[0] == 'HKEY_CURRENT_USER':
```

```
            key = RegOpenKey(HKEY_CURRENT_USER,name[1], 0, KEY_READ)
        elif name[0] == 'HKEY_CLASSES_ROOT':
            key = RegOpenKey(HKEY_CLASSES_ROOT,name[1], 0, KEY_READ)
        elif name[0] == 'HKEY_CURRENT_CONFIG':
            key = RegOpenKey(HKEY_CURRENT_CONFIG,name[1], 0, KEY_READ)
        elif name[0] == 'HKEY_USERS':
            key = RegOpenKey(HKEY_USERS,name[1], 0, KEY_READ)
        else:
            print('err,no key named %s' % name[0])

        # 查询项的项值数目
        info = RegQueryInfoKey(key)
        # 遍历项值获得项值数据
        for i in range(0,info[1]):
            ValueName = RegEnumValue(key, i)
            #调整项值名称长度，使输出更好看
            print(str.ljust(ValueName[0],20),ValueName[1])
        # 关闭打开的项
        RegCloseKey(key)
    except:
        pass

# 因为 GetValues 函数比较通用，可以在其他脚本中调用
# 这里先检查脚本是否被其他脚本调用
if __name__ == '__main__':
    # 因为要要检查的项较多，故将其放在列表中，便于增减
    KeyNames = ['HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\
CurrentVersion\\Run',\
                'HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\
CurrentVersion\\RunOnce',\
                'HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\
CurrentVersion\\RunOnceEx',\
                'HKEY_CURRENT_USER\\Software\\Microsoft\\Windows\\
CurrentVersion\\Run',\
                'HKEY_CURRENT_USER\\Software\\Microsoft\\Windows\\
CurrentVersion\\RunOnce']
    # 遍历列表，调用 GetValues 函数，输出项值
    for KeyName in KeyNames:
        print(KeyName)
        GetValues(KeyName)
```

10.1.4 修改 IE

IE 浏览器的相关设置也是保存在注册表中的，只要通过设置注册表中相应的项值，就可以修改 IE 的主页、标题栏等。与 IE 设置相关的几个注册表项如下。

- HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Main\Start Page 项保存的是 IE 的主页地址。
- HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Main\Window Title 项保存的是 IE 的标题栏。
- HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Main\Search Page 项保存的是 IE 默认的搜索页。

为了打造一个个性化的 IE，可以编写一个 Python 脚本，利用 10.1.3 节的知识将其设置为开机自动运行，然后通过脚本获得当前的日期，并随机选择预先设置好的个性标题，这样就可以打造出每天都不同的个性化 IE 了。还可以使用 Python 的网络编程功能获取天气预报网站上的天气情况，将其添加到 IE 的标题栏。这里暂时先不实现这个功能，在完成后面章节的学习后，再将获取天气情况的功能添加到该脚本中。

首先编写将脚本设置为开机运行的 Add2AtuoRun.py，其代码如下。

```
#  -*- coding: utf-8 -*-
#  file: Add2AutoRun.py
#
import win32api
import win32con
name = 'SetIE'                                                    # 要添加的项值名称
path = 'C:\\SetIE.py'                                             # 要添加的 Python 脚本的
路径
KeyName = 'Software\\Microsoft\\Windows\\CurrentVersion\\Run' # 注册表项名
# 异常处理
try:
    # 打开注册表项
    key = win32api.RegOpenKey(win32con.HKEY_CURRENT_USER,\
                          KeyName,\
                          0,\
                          win32con.KEY_ALL_ACCESS)
    win32api.RegSetValueEx(key, name, 0, win32con.REG_SZ, path)     # 设置项值
    win32api.RegCloseKey(key)                                       # 关闭注册表
except:
    print( 'error')
print('added that!')
```

如果要使其他的脚本也能开机自动运行，则只需修改其变量 name 及 path，将它们分别设置为启动项目的名字和脚本所在的路径。接着编写以下代码，实现修改 IE 主页、打造个性化 IE 标题栏等功能。

```
#  -*- coding: utf-8 -*-
#  file: SetIE.py
#
import datetime
import string
import win32api
import win32con
keyname = 'Software\Microsoft\Internet Explorer\Main'  # 要修改的注册表项
page = 'www.python.org'                                # 要设置为主页的网址
today = datetime.date.today()                          # 获取当前日期
# 将日期格式化为 xxxx 年 xx 月 xx 日的形式
title = today.strftime('%Y')+'年'+today.strftime('%m')+'月
'+today.strftime('%d')+'日'
# 异常处理
try:
    # 打开注册表项，获得句柄
```

```
    key = win32api.RegOpenKey(win32con.HKEY_CURRENT_USER, keyname, 0, win32con.
KEY_ALL_ACCESS)
    # 读取"Start Page"的项值数据
    StartPage = win32api.RegQueryValueEx(key, 'Start Page')
except:
    print('error')
else:
    # 判断主页是否为要修改的主页，如果不是则修改
    if StartPage[0] != page:
        win32api.RegSetValueEx(key, 'Start Page', 0, win32con.REG_SZ, page)
    # 设置 IE 的标题栏为 xxxx 年 xx 月 xx 日
    win32api.RegSetValueEx(key, 'Window Title', 0, win32con.REG_SZ, title)
    win32api.RegCloseKey(key)                                    # 关闭注册表
```

脚本运行后，如果电脑中的 IE 浏览器是 IE9 之前的版本，则可以看到 IE 浏览器的标题栏被设置为 xxxx 年 xx 月 xx 日的形式，如图 10-2 所示。如果 IE 浏览器是 IE9 及之后的版本，则显示效果如图 10-3 所示（在 IE9、IE10 等版本中没有标题栏，因此将不会显示日期，但是设置的主页仍然有效）。



图 10-2 修改后的含 xxxx 年 xx 月 xx 日的 IE 界面



图 10-3 修改后的 IE9 界面

另外，需要注意的是，在 Windows 7 中修改注册表时需要有管理员权限，否则执行以上脚本将显示权限不足的错误，用 Administrator 用户操作则不会出现这类错误。

10.2 文件和目录

在计算机系统中进行操作时，免不了要与文件和目录打交道。对于一些比较烦琐的文件和目录操作，可以使用 Python 提供的 os 模块来进行。os 模块中包含很多操作文件和目录的函数，可以方便地进行重命名文件、添加/删除目录、复制目录/文件等操作。

10.2.1 文件目录常用函数

在进行文件和目录操作时，一般会用到以下几种操作。

1. 获得当前路径

在 Python 中，可以使用 os.getcwd()函数获得当前路径。其原型如下。

```
os.getcwd()
```

该函数不需要传递参数，它返回的是当前的目录。需要说明的是，当前目录并不是指脚本所在的目录，而是所运行脚本的目录。例如，在 PythonWin 中输入以下脚本。

```
>>> import os
>>> print('Current directory is ',os.getcwd())
Current directory is  C:\Python32        #这里是 Python 的安装目录
```

如果将上述内容写入 pwd.py，假设 pwd.py 位于“E:\Python\第 10 章”目录，运行 Windows 命令行窗口，进入“E:\Python\第 10 章”目录，输入 pwd.py，则输出如下。

```
E:\Python\第 10 章>pwd.py
Current directory is  E:\Python\第 10 章
```

2. 获得目录中的内容

在 Python 中，可以使用 os.listdir()函数获得指定目录中的内容。其原型如下。

```
os.listdir(path)
```

其参数含义如下。

- path　　要获得内容的目录的路径。

以下实例获得当前目录的内容。

```
>>> import os
>>> os.listdir(os.getcwd())          # 获得当前目录中的内容(以下结果在 C:\Python32 目录中的内容)
['DLLs', 'Doc', 'include', 'kages', 'Lib', 'libs', 'LICENSE.txt', 'NEWS.txt',
'Pyrex-wininst.log', 'python.exe', 'pythonw.exe', 'pywin32-wininst.log',
'README.txt', 'RemovePyrex.exe', 'Removepywin32.exe', 'Scripts', 'src', 'tcl',
'Tools', 'w9xpopen.exe']
```

3. 创建目录

在 Python 中可以使用 os.mkdir()函数创建目录，其原型如下。

```
os.mkdir(path)
```

其参数含义如下。

◆ path　　要创建目录的路径。

下面的代码将在“E:\Python\第 10 章”目录下创建 temp 目录。

```
>>> import os
>>> os.mkdir('E:\\Python\\第 10 章\\temp')        # 使用 os.mkdir 创建目录
```

4. 删除目录

在 Python 中可以使用 os.rmdir()函数删除目录，其原型如下。

```
os.rmdir(path)
```

其参数含义如下。

◆ path　　要删除的目录的路径。

下面的代码将删除“E:\Python\第 10 章\temp”目录。

```
>>> import os
>>> os.rmdir('E:\\Python\\第 10 章\\temp')        # 删除目录
```

需要说明的是，使用 os.rmdir 删除的目录必须为空目录，否则函数出错。如果删除的目录不存在，则也会报错。

5. 判断是否是目录

在 Python 中，可以使用 os.path.isdir()函数判断某一路径是否为目录，其函数原型如下。

```
os.path.isdir(path)
```

其参数含义如下。

◆ path　　要进行判断的路径。

下面的代码可以判断 E:\book\temp 是否为目录。

```
>>> import os
>>> os.path.isdir('E:\\Python')     # 判断 E:\Python 是否为目录
True                    # 为 True 则表示 E:\Python 是目录，为 False 则表示不是目录或不存在
```

6. 判断是否为文件

在 Python 中，可以使用 os.path.isfile()函数判断某一路径是否为文件，其函数原型如下。

```
os.path.isfile(path)
```

其参数含义如下。

◆ path　　要进行判断的路径。

下面的代码可以判断 E:\book\temp 是否为文件。

```
>>> import os
>>> os.path.isfile('E:\\Python')    # 判断是否为文件
False                               # 为 False 则表示 E:\Python 不是文件
```

10.2.2 批量重命名

在日常工作中经常会遇到这样的情况，需要将某个文件夹下的文件按照一定的规则重新命名。如果用手工方式逐个文件进行重命名的话，则需要耗费大量的时间，而且操作过程容易出错。在学习了 Python 以后，完全可以写一个简单的脚本完成这样的工作。

```
import os
perfix = 'Python'                           # perfix 为重命名后的文件起始字符
length = 2                                  # length 为除去 perfix 后，文件名要达到的长度
base = 1                                    # 文件名的起始数
format = 'mdb'                              # 文件的后缀名
# 函数 PadLeft 将文件名补全到指定长度
# str 为要补全的字符
# num 为要达到的长度
# padstr 未达到长度所添加的字符
def PadLeft(str , num , padstr):
    stringlength = len (str)
    n = num - stringlength
    if n >=0 :
        str=padstr * n + str
    return str
# 为了避免误操作，这里先提示用户
Print('the files in %s will be renamed' % os.getcwd())
input = input('press y to continue\n') # 获取用户输入
if input.lower() != 'y':                    # 判断用户输入，以决定是否执行重命名操作
    exit()
filenames = os.listdir(os.curdir)           # 获得当前目录中的内容
# 基数减 1，为了使下面的 i = i + 1 在第一次执行时等于基数
i = base - 1
for filename in filenames:                  # 遍历目录中的内容，进行重命名操作
    i = i+1
    # 判断当前路径是否为文件，并且不是 "rename.py"
    if filename != "rename.py" and os.path.isfile(filename):
        name = str(i)                       # 将 i 转换成字符
        name = PadLeft(name,length,'0')# 将 name 补全到指定长度
        t = filename.split('.')             # 分割文件名，以检查其是否是所要修改的类型
        m = len(t)
        if format == '':                    # 如果未指定文件类型，则更改当前目录中的所有文件
            os.rename(filename,perfix+name+'.'+t[m-1])
        else:                               # 否则只修改指定类型
```

```
            if t[m-1] == format:
                os.rename(filename,perfix+name+'.'+t[m-1])
            else:
                i = i - 1                    # 保证 i 连续
    else:
        i = i - 1                            # 保证 i 连续
```

10.2.3 代码框架生成器

在编写 Python 脚本代码时，为了使脚本更具可读性，往往需要在代码文件开始处添加一些注释。通常在脚本头部添加基本的说明内容，如作者、文件名、日期、用途、版权说明，以及所需要使用的模块等信息。这样，不仅便于保存脚本，而且也便于交流。

但是，由于这些说明信息大部分都是相同的，因此，如果每编写一个脚本就依次重复添加这些信息，则不免有些麻烦。其实，用 Python 可以编写自动生成这些信息的代码，以下代码就实现了一个简单的代码框架生成器。

```
# -*- coding:utf-8 -*-
# file: MakeCode.py
#
import os
import sys
import datetime
# python 脚本模板
py = '''#-------------------------------------------------------
# TO:
#-------------------------------------------------------
# BY:
#-------------------------------------------------------
'''
# c 模板
c = ''' *-------------------------------------------------------
 * TO:
 *-------------------------------------------------------
 * BY:
 *-------------------------------------------------------
'''
if os.path.isfile(sys.argv[1]):          # 判断要创建的文件是否存在，如果存在则退出脚本
    print('%s already exist!' % sys.argv[1])
    sys.exit()
file = open(sys.argv[1], 'w')            # 创建文件
today = datetime.date.today()            # 获得当前日期，并格式化为 xxxx-xx-xx 的形式
date = today.strftime('%Y')+'-'+today.strftime('%m')+'-'+today.strftime('%d')
filetypes = str.split(sys.argv[1],'.') # 判断将创建的文件是什么类型，以便对其分别处理
length = len(filetypes)
filetype = filetypes[length - 1]
if filetype == 'py':
    print('use python mode')
    file.writelines('# -*- coding:utf-8 -*-')
    file.write('\n')
    file.writelines('# File: ' + sys.argv[1])
    file.write('\n')
    file.write(py)
    file.write('# Date: ' + date)
    file.write('\n')
```

```
    file.write('#-------------------------------------------------------')
elif filetype == 'c' or filetype == 'cpp':
    print('use c mode')
    file.writelines('/*')
    file.write('\n')
    file.writelines(' *-------------------------------------------------------')
    file.write('\n')
    file.writelines(' * File: ' + sys.argv[1])
    file.write('\n')
    file.write(c)
    file.write(' * Date: ' + date)
    file.write('\n')
    file.write(' *-------------------------------------------------------')
    file.write('\n')
    file.write(' */ \n')
else:
    print('just create %s' % sys.argv[1])
file.close()                                    # 关闭文件
```

编写好上述脚本后，就可以随时运行来生成脚本框架了。例如，若要在当前目录中生成一个名为 test.py 的脚本框架，则可以在命令窗口输入以下命令：

```
E:\Python\第 10 章>MakeCode.py test.py
```

执行上述命令后，在当前目录中将生成一个名为 test.py 的文件，打开该文件可看到具体内容如下。

```
# -*- coding:utf-8 -*-
# File: test.py
#-------------------------------------------------------
# TO:
#-------------------------------------------------------
# BY:
#-------------------------------------------------------
# Date: 2013-11-13
#-------------------------------------------------------
```

从上面的代码可看到，生成的脚本框架包含了编码注释、文件名、一些附注说明以及生成日期等信息。当然，如果是在做一个比较复杂的项目，则可能每个文件头还需要包含一些其他信息，也可由 MakeCode.py 脚本来生成。

10.3 生成可执行文件

使用 Python 固然方便，但必须在要执行 Python 脚本的计算机中安装 Python 解释器。如果当前计算机没有安装 Python 解释器，而又需要运行脚本，则可考虑将 Python 脚本打包封装成 Windows 可执行文件，这样就不需要 Python 解释器了。

最常用的是 py2exe 将 Python 脚本文件打包封装为可执行文件，不过，这个工具目前还不支持 Python 3。对于 Python 3 而言，则需要借助另一个工具软件：cx_freeze。

10.3.1 安装 py2exe

首先来看 py2exe。py2exe 可以从其官方网站（http://www.py2exe.org/）下载，下载时应注意根据所安装的版本选择相应的安装程序（目前只支持到 Python 2.7）。

py2exe 的安装十分简单，其安装程序会自动搜索 Python 的安装目录。双击安装程序，单击【下一步】按钮，如果安装程序能搜索到 Python 的安装路径，则会出现如图 10-4 所示界面。只需要一直单击【下一步】按钮，即可完成安装。安装完成后可以在 Python Shell 中输入以下所示代码。

```
>>> import py2exe
# 如果有以下输出则表示未安装成功，如果没有输出，则表示 py2exe 模块已经安装好
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
ImportError: No module named py2exe
```



图 10-4 安装 py2exe

10.3.2 使用 py2exe 生成可执行文件

要使用 py2exe，首先要编写一个编译脚本（如编写一个名为 setup.py 的脚本），然后在 Python 中运行编译脚本 setup.py，即可将需要封装的其他 Python 脚本编译成可执行文件。

例如，有一个名为 MessageBox.py 的脚本，其代码如下。

```
#file: MessageBox.py
import win32api
import win32con
win32api.MessageBox(0, 'hi!', 'Python', win32con.MB_OK)
```

若要将 MessageBox.py 脚本编译成可执行文件，则需首先编写以下编译脚本。

```
#file: setup.py
import distutils
import py2exe
distutils.core.setup(windows=['MessageBox.py'])
```

在 Windows 的 cmd 窗口中输入 “setup.py py2exe”，将得到以下输出。

```
running py2exe
*** searching for required modules ***
*** parsing results ***
```

```
creating python loader for extension 'win32api'
creating python loader for extension 'unicodedata'
creating python loader for extension 'bz2'
*** finding dlls needed ***
*** create binaries ***
*** byte compile python files ***
********************************************************************************
部分输出省略
********************************************************************************
byte-compiling E:\book\code\py2exe\build\bdist.win32\winexe\temp\unicodedata.py
to unicodedata.pyc
byte-compiling E:\book\code\py2exe\build\bdist.win32\winexe\temp\win32api.py to
win32api.pyc
*** copy extensions ***
*** copy dlls ***
copying D:\Python25\lib\site-packages\py2exe\run_w.exe -> E:\book\code\py2exe\di
st\MessageBox.exe

*** binary dependencies ***
Your executable(s) also depend on these dlls which are not included,
you may or may not need to distribute them.

Make sure you have the license if you distribute any of them, and
make sure you don't distribute files belonging to the operating system.

   OLEAUT32.dll - C:\WINDOWS\system32\OLEAUT32.dll
   USER32.dll - C:\WINDOWS\system32\USER32.dll
   SHELL32.dll - C:\WINDOWS\system32\SHELL32.dll
   ole32.dll - C:\WINDOWS\system32\ole32.dll
   ADVAPI32.dll - C:\WINDOWS\system32\ADVAPI32.dll
   VERSION.dll - C:\WINDOWS\system32\VERSION.dll
   KERNEL32.dll - C:\WINDOWS\system32\KERNEL32.dll
```

运行完编译脚本以后，会在当前文件夹下生成 dist 和 build 两个目录。其中，dist 目录中是编译生成的文件。如果要在其他未安装 Python 的机器上运行编译好的程序，则只需将 dist 目录复制到其他机器上即可。双击运行 MessageBox.exe，如图 10-5 所示。



图 10-5　运行 MessageBox.exe

在 setup.py 中除了导入必需的模块外，只有一条语句。

```
distutils.core.setup(windows=['MessageBox.py'])
```

方括号中就是要编译的脚本名，其前边的 windows 表示将其编译成 GUI 程序。如果要编译命令行界面的可执行文件，则只需将 windows 改为 console。重新编译 MessageBox.py 后，双击运行如图 10-6 所示。可以看到运行程序后多了一个命令行窗口。另外，如果需要将脚本编译成 Windows 服务，则可以使用 service 选项。



图 10-6 重新编译的 MessageBox.exe

10.3.3 使用 cx_freeze 生成可执行文件

与使用 py2exe 类似，首先需将 cx_freeze 下载到本机并安装后才能使用。下载地址为：http://sourceforge.net/projects/cx-freeze/files/，下载时需注意选择对应的 Python 版本，以及操作系统平台。例如，在 Windows 中的 Python 3.2.5 中使用，则可下载安装文件 cx_Freeze-4.3.1.win32-py3.2.msi。

安装很简单，双击下载的安装文件，然后根据向导单击鼠标即可完成安装。安装完成后，在 C:\Python32\Scripts 目录中可以看到 cx_freeze 的相关文件。在命令窗口切换到 C:\Python32\Scripts 目录，输入以下命令，可看到如图 10-7 所示效果，表示 cx_freeze 安装成功。

```
C:\Python32\Scripts>cxfreeze -h
```

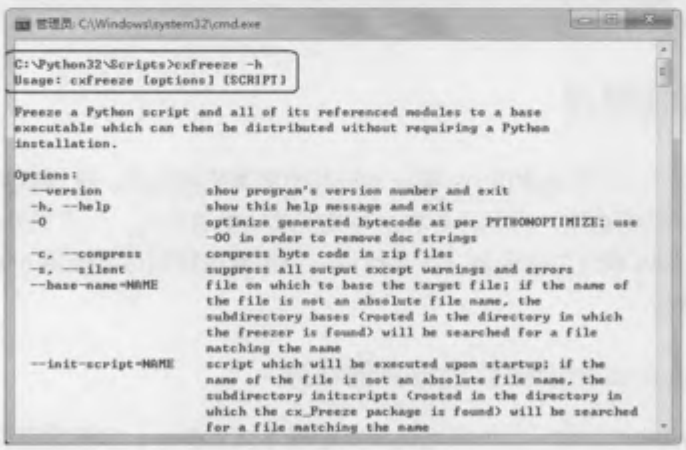


图 10-7 测试 cx_freeze

cx_freeze 安装成功后，接下来就可以进行 Python 脚本的编译操作了。仍以 10.3.2 节操作的 MessageBox.py 为例，演示在 Python3 中使用 cx_freeze 把脚本编译为 exe 文件的方法。

step 1 在命令窗口切换到“E:\Python\第 10 章”目录（需打包文件所在目录）。

step 2 输入以下命令：

```
E:\Python\第 10 章> >c:\python32\Scripts\cxfreeze  MessageBox.py  --target-dir dist
```

其中，MessageBox.py 是需要编译的脚本文件，dist 是目标文件夹，打包后会生成 dist 目录，在这个目录中生成编译后的可执行文件。

执行上述命令后，将显示如图 10-8 所示的输出。



图 10-8　使用 cx_freeze 编译

最后在当前目录下的 dist 目录中生成可执行的 exe 文件，用鼠标双击该文件，可看到与图 10-6 类似的界面。

如果想得到如图 10-5 所示的界面（即没有控制台窗口作背景），则在编译时还需添加一个 base-name 的参数，具体命令如下：

```
E:\Python\第 10 章> >c:\python32\Scripts\cxfreeze  MessageBox.py  --target-dir
dist  --base-name=Win32GUI
```

10.4　运行其他程序

在 Python 中，可以方便地使用 os 模块运行其他脚本或者程序。这样就可以在脚本中直接使用其他脚本或程序提供的功能，而不必再次编写实现该功能的代码。为了更好地控制运行的进程，可以使用 win32process 模块中的函数，如果想进一步控制进程则可以使用 ctype 模块，直接调用 kernel32.dll 中的函数。

10.4.1　使用 os.system()函数运行其他程序

os 模块中的 system()函数可以方便地运行其他程序或者脚本。其函数原型如下。

```
os.system(command)
```

其参数含义为。

- command　要执行的命令，相当于在 Windows 的 cmd 窗口中输入的命令。如果要向程序或者脚本传递参数，可以使用空格分割程序及多个参数。

以下代码可以实现通过 os.system()函数打开系统的记事本程序。

```
>>> import os
# 使用os.system()函数打开记事本程序
>>> os.system('notepad')
0 # 关闭记事本后的返回值
# 向记事本传递参数，打开python.txt文件
>>> os.system('notepad python.txt')
0
```

10.4.2 使用ShellExecute函数运行其他程序

除了使用os模块中的os.system()函数外，还可以使用win32api模块中的ShellExecute()函数来运行其他程序。其函数如下。

```
ShellExecute(hwnd, op , file , params , dir , bShow )
```

其参数含义如下。

- hwnd　父窗口的句柄，如果没有父窗口，则为0。
- op　要进行的操作，为“open”、“print”或者为空。
- file　要运行的程序，或者打开的脚本。
- params　要向程序传递的参数，如果打开的是文件则为空。
- dir　程序初始化的目录。
- bShow　是否显示窗口。

以下代码是使用ShellExecute()函数运行其他程序。

```
>>> import win32api
# 打开记事本程序，在后台运行，即显示记事本程序的窗口
>>> win32api.ShellExecute(0, 'open', 'notepad.exe', '','',0)
42
# 打开记事本程序，在前台运行
>>> win32api.ShellExecute(0, 'open', 'notepad.exe', '','',1)
42
# 向记事本传递参数，打开python.txt
>>> win32api.ShellExecute(0, 'open', 'notepad.exe', 'python.txt','',1)
42
# 在默认浏览器中打开http://www.python.org网站
>>> win32api.ShellExecute(0, 'open', 'http://www.python.org', '','',1)
42
# 在默认的媒体播放器中播放E:\song.wma
>>> win32api.ShellExecute(0, 'open', 'E:\\song.wma', '','',1)
42
# 运行位于E:\book\code目录中的MessageBox.py脚本
>>> win32api.ShellExecute(0, 'open', 'E:\\Python\\第10章\\MessageBox.py',
'','',1)
42
```

可以看出，使用ShellExecute函数，就相当于在资源管理器中双击文件图标，系统会打开相应的应用程序执行操作。

10.4.3 使用 CreateProcess 函数运行其他程序

为了便于控制通过脚本运行的程序，可以使用 win32process 模块中的 CreateProcess()函数创建一个运行相应程序的进程。其函数原型如下。

```
CreateProcess(appName, commandLine , processAttributes , threadAttributes ,
bInheritHandles , dwCreationFlags , newEnvironment , currentDirectory ,
startupinfo )
```

其参数含义如下。

- appName　　可执行文件名。
- commandLine　　命令行参数。
- processAttributes　　进程安全属性，如果为 None 则为默认的安全属性。
- threadAttributes　　线程安全属性，如果为 None 则为默认的安全属性。
- bInheritHandles　　继承标志。
- dwCreationFlags　　创建标志。
- newEnvironment　　创建进程的环境变量。
- currentDirectory　　进程的当前目录。
- startupinfo　　创建进程的属性。

以下代码是使用 win32process.CreateProcess 函数运行记事本程序。

```
>>> import win32process
>>> win32process.CreateProcess('c:\\windows\\notepad.exe', '', None , None ,
0 ,win32process. CREATE_NO_WINDOW , None , None ,win32process.STARTUPINFO())
(<PyHANDLE:584>, <PyHANDLE:600>, 280, 3076) # 函数返回进程句柄、线程句柄、进程 ID 以及
线程 ID
```

有了已创建进程的句柄就可以使用 win32process.TerminateProcess 函数结束进程，或者使用 win32event.WaitForSingleObject 等待创建的线程结束。其函数原型如下。

```
TerminateProcess(handle, exitCode)
WaitForSingleObject(handle, milliseconds )
```

对于 TerminateProcess，其参数含义如下。

- handle　　为要操作的进程句柄。
- exitCode　　进程退出代码。

对于 WaitForSingleObject，其参数含义如下。

- handle　　为要操作的进程句柄。
- milliseconds　　等待的时间，如果为-1 则一直等待。

以下代码用于创建进程，然后对创建的进程进行相应的操作。

```
>>> import win32process
```

```
# 打开记事本程序，获得其句柄
>>> handle = win32process.CreateProcess('c:\\windows\\notepad.exe', '', None ,
None , 0 ,win32process. CREATE_NO_WINDOW , None , None ,win32process.STARTUPINFO())
# 使用 TerminateProcess 函数终止记事本程序
>>> win32process.TerminateProcess(handle[0],0)
# 导入 win32event 模块
>>> import win32event
# 创建进程获得句柄
>>> handle = win32process.CreateProcess('c:\\windows\\notepad.exe', '', None ,
None , 0 ,win32process. CREATE_NO_WINDOW , None , None ,win32process.STARTUPINFO())
# 等待进程结束
>>> win32event.WaitForSingleObject(handle[0], -1)
0  # 进程结束的返回值
```

10.4.4 使用 ctypes 调用 kernel32.dll 中的函数

使用 ctypes 模块可以让 Python 调用位于动态链接库中的函数。在 Python 3.2 版中包含有 ctypes 模块，如果使用其他未包含 ctypes 模块的 Python 版本，则可以到 http://python.net/crew/theller/ctypes/网站下载安装 ctypes 模块。ctypes 模块适用于 Python 2.3 及以上版本。

1. ctypes 模块简介

ctypes 模块为 Python 提供了调用动态链接库中函数的功能。使用 ctypes 模块可以方便地调用由 C 语言编写的动态链接库，并向其传递参数。ctypes 模块定义了 C 语言中的基本数据类型，并且可以实现 C 语言中的结构体和联合体。ctypes 模块可以工作在 Windows、Windows CE、Mac OS X、Linux、Solaris、FreeBSD、OpenBSD 等平台上，基本上实现了跨平台。

以下代码是使用 ctypes 模块在 Windows 下直接调用 user32.dll 中的 MessageBoxA 函数。运行后如图 10-9 所示。

```
>>> from ctypes import *
>>> user32 = windll.LoadLibrary('user32.dll')                    # 加载动态链接库
>>> user32.MessageBoxA(0, str.encode('Ctypes is cool!'),str.encode('Ctypes'), 0)
    # 调用 MessageBoxA 函数
1
```



图 10-9 使用 ctypes 模块

2. 数据类型与结构体

ctypes 模块中含有的基本类型与 C 语言类似，表 10-2 中列出了几个基本的数据类型的对照。

表 10-2 数据类型对照

Ctypes 数据类型	C 数据类型
c_char	char
c_short	short

续表

Ctypes 数据类型	C 数据类型
c_int	int
c_long	long
c_float	float
c_double	double
c_void_p	void *

在 Python 中要运行 C 语言的结构体，需要使用类。PROCESS_INFORMATION 的原型如下。

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION,
 *LPPROCESS_INFORMATION;
```

在 Python 中，使用 ctypes 模块可以运行 Windows 中的 PROCESS_INFORMATION 结构体，但需编写如下所示的类来实现。

```
class _PROCESS_INFORMATION(Structure):
    _fields_ = [('hProcess', c_void_p),
            ('hThread', c_void_p),
            ('dwProcessId', c_ulong),
            ('dwThreadId', c_ulong)]
```

要声明一个 PROCESS_INFORMATION 类型的数据，只需使用以下语句即可。

```
ProcessInfo = _PROCESS_INFORMATION()
```

如果在函数中要向结构体中赋值，可以使用 byref，byref 相当于 C 语言中的“&”。

10.5 本章小结

本章介绍了 Python 操作系统编程方面的内容，如通过 Python 脚本操作 Windows 的注册表，通过 Python 脚本操作文件和目录。接着还介绍了使用 py2exe 模块将 Python 2 脚本打包为 exe 可执行程序的方法。对于 Python 3 的脚本，可使用 cx_freeze 模块打包为 exe 可执行文件。本章最后还介绍了在 Python 中运行其他程序的方法。

从下一章开始，将用 5 章的篇幅介绍 Python 的图形用户界面设计。

◆ ◆ ◆

第 11 章 使用 PythonWin 编写 GUI

本章包括

- Windows GUI 编程概述
- 使用 MFC 创建窗口
- 创建菜单
- 使用 Windows API 创建窗口
- 创建对话框
- 处理菜单消息

GUI 是图形用户界面（Graphical User Interface）的英文简称，是指采用图形方式显示的计算机操作用户界面。Windows 的图形用户界面非常方便用户操作，因此，Windows 操作系统得到了广大个人计算机用户的欢迎。在 Python 中，也可以编写美观的 GUI 界面，通过使用 PythonWin 中的 win32gui 和 win32ui 模块可以调用 Windows API 来创建 GUI 界面。

11.1 Windows GUI 编程概述

在 Windows 操作系统下，可以直接使用 Windows API 创建 GUI 程序。由于使用 Windows API 的操作较为烦琐，因此，Windows 提供了 MFC 类库对 Windows API 进行封装。使用 MFC 类库创建 GUI 程序比直接调用 Windows API 函数要方便得多。但是，MFC 类库相当庞大，使用较为复杂，不容易掌握。

下面先来介绍直接调用 Windows API 函数创建窗口的操作，接着演示通过 MFC 创建窗口的过程。

11.1.1 使用 Windows API 创建窗口

在 Python 中，使用 PythonWin 提供的 Windows API 创建窗口，与在 VC++6.0 中使用 Windows API 编写 GUI 的过程相同。

1. 创建窗口

使用 Windows API 创建窗口，首先需要使用 win32gui 模块中的 RegisterClass 函数注册窗口类，定义消息回调函数，然后使用 win32gui 模块中的 CreateWindow 函数创建并显示窗口。

以下 WinGUI.py 脚本是使用 PythonWin 提供的 Windows API 创建一个简单的窗口。

```
# -*- coding:utf-8 -*-
# file: WinGUI.py
#
import win32gui
from win32con import *
# 窗口消息处理函数
def WndProc(hwnd, msg, wParam, lParam):
    if msg == WM_PAINT:
        hdc,ps = win32gui.BeginPaint(hwnd)
```

```
        rect = win32gui.GetClientRect(hwnd)
        win32gui.DrawText(hdc,
                'GUI Python',
                len('GUI Python'),
                rect,
                DT_SINGLELINE|DT_CENTER|DT_VCENTER)
        win32gui.EndPaint(hwnd,ps)
    if msg == WM_DESTROY:
        win32gui.PostQuitMessage(0)
    return win32gui.DefWindowProc(hwnd, msg, wParam, lParam)
# 生成窗口类，并对相关项赋值
wc = win32gui.WNDCLASS()
wc.hbrBackground = COLOR_BTNFACE + 1
wc.hCursor = win32gui.LoadCursor(0, IDC_ARROW)
wc.hIcon = win32gui.LoadIcon(0, IDI_APPLICATION)
wc.lpszClassName = "Python on Windows"
wc.lpfnWndProc = WndProc
# 注册窗口类
reg = win32gui.RegisterClass(wc)
# 创建窗口
hwnd = win32gui.CreateWindow(
          reg, 'Python', WS_OVERLAPPEDWINDOW,
          CW_USEDEFAULT, CW_USEDEFAULT,
          CW_USEDEFAULT, CW_USEDEFAULT,
          0,0, 0, None)
# 显示窗口
win32gui.ShowWindow(hwnd, SW_SHOWNORMAL)
win32gui.UpdateWindow(hwnd)
# 进入消息循环，直至窗口结束
win32gui.PumpMessages()
```

运行 WinGUI.py 脚本后会创建如图 11-1 所示的窗口。



图 11-1　Python GUI 窗口

2. 脚本说明

在上面的脚本中主要使用了创建窗口、显示窗口、输出文字等函数。其中函数 win32gui.CreateWindow 用于创建窗口，它返回创建窗口的句柄。其参数原型如下。

```
CreateWindow(className, windowTitle , style , x , y , width , height , parent ,
menu , hinstance , reserved )
```

其参数含义如下。

- className　　注册的窗口类名。
- windowTitle　　窗口标题名。
- style　　窗口样式。
- x　　窗口左上角 X 坐标。
- y　　窗口左上角 Y 坐标。
- width　　窗口的宽度。
- height　　窗口的高度。
- parent　　父窗口句柄，如果没有则设置为 0。
- menu　　菜单 ID，如果没有则设置为 0。
- hinstance　　该值在 Windows XP 下被忽略，设置为 0。
- reserved　　保留，应为 None。

当使用 win32gui.CreateWindow 创建窗口后，还要使用 win32gui.ShowWindow 函数才能显示窗口。其参数原型如下。

```
ShowWindow(hwnd, cmdShow )
```

其参数含义如下。

- hwnd　　窗口句柄，即 win32gui.CreateWindow 函数的返回值。
- cmdShow　　指定窗口的显示状态。

在 win32gui.ShowWindow 函数之后，使用了 win32gui.UpdateWindow 函数，其作用是发送 WM_PAINT 消息，通知系统刷新窗口。其参数原型如下。

```
UpdateWindow(hwnd)
```

其参数含义如下。

- hwnd　　窗口句柄，即 win32gui.CreateWindow 函数的返回值。

在脚本中还定义了如下所示的函数。

```
def WndProc(hwnd, msg, wParam, lParam)
```

该函数作为 Windows 消息处理的回调函数。在脚本中将其赋值给窗口类的 wc.lpfnWndProc。当窗口创建以后，使用 win32gui.PumpMessages()进入无限消息循环，处理窗口消息。窗口消息首先传递给 WndProc，在 WndProc 中可以定义相应消息的处理过程。在 WinGUI.py 中处理了 WM_PAINT 和 WM_DESTROY 消息。其中，在 WM_PAINT 消息中向窗口输出“GUI Python”，在 WM_DESTROY 消息中向窗口发送退出消息，对于其他的消息，则调用默认的消息处理函数 win32gui.DefWindowProc 进行处理。

11.1.2 使用 MFC 创建窗口

MFC 对基本的 Windows API 函数进行了封装。使用 MFC 创建窗口的大致过程是：首先，创建窗口类；然后，重载消息处理方法；最后，类实例化，创建窗口。

当窗口创建后，需要调用 RunModalLoop 方法进入窗口消息循环，否则窗口将立刻关闭。另外需要重载 OnClose 方法，使用 EndModalLoop 方法结束消息循环。

下面所示的 MFCGUI.py 脚本是使用 MFC 创建一个简单的窗口。

```
# -*- coding:utf-8 -*-
# file: MFCGUI.py
#
import win32ui
import win32api
from win32con import *
from pywin.mfc import window
# 定义窗口类
class MyWnd(window.Wnd):
    def __init__ (self):
        window.Wnd.__init__(self, win32ui.CreateWnd ())
        self._obj_.CreateWindowEx(WS_EX_CLIENTEDGE, \
                win32ui.RegisterWndClass(0, 0, COLOR_WINDOW + 1), \
                'MFC GUI', WS_OVERLAPPEDWINDOW, \
                (100, 100, 400, 300), None, 0, None)
    # 重载 OnClose 方法
    def OnClose(self):
        self.EndModalLoop(0)
    # 重载 OnPaint 方法，在窗口中输出 "MFC GUI"
    def OnPaint(self):
        dc,ps = self.BeginPaint()
        dc.DrawText('MFC GUI',
            self.GetClientRect(),
            DT_SINGLELINE | DT_CENTER | DT_VCENTER)
        self.EndPaint(ps)
w = MyWnd()                                          # 生成窗口对象
w.ShowWindow()                                       # 显示窗口
w.UpdateWindow()                                     # 刷新窗口
w.RunModalLoop(1)                                    # 进入消息循环
```

运行 MFCGUI.py 脚本后，将创建如图 11-2 所示的窗口。



图 11-2 使用 MFC 创建 GUI 窗口

11.2 创建对话框

在多数 GUI 程序中都会使用到对话框。对话框可用来与用户交互，以收集程序运行所需要的

相关信息。对于某些简单的 GUI 程序而言，其本身就是一个对话框。

创建对话框比创建窗口容易，如果不希望脚本太复杂，则可以考虑在脚本中使用对话框来代替窗口。

11.2.1 创建对话框

通过重载 pywin.mfc 模块中的 Dialog，可以创建简单的对话框。

对话框的创建过程与窗口的创建过程类似，但相对要简单些。下面所示的 SimpleDialog.py 脚本，是通过重载 pywin.mfc 模块中的 dialog.Dialog 来创建一个简单的对话框。

```
# -*- coding:utf-8 -*-
# file: SimpleDialog.py
#
import win32ui                                  # 导入 win32ui 模块
import win32con                                 # 导入 win32con 模块
from pywin.mfc import dialog                    # 导入 pywin.mfc 中的 dialog 模块
class MyDialog(dialog.Dialog):                  # 通过继承 dialog.Dialog 生成对话框类
    def OnInitDialog(self):                     # 重载初始化函数
        dialog.Dialog.OnInitDialog(self)        # 调用父类的初始化函数
style = (win32con.DS_MODALFRAME     |           # 定义对话框样式
        win32con.WS_POPUP       |
        win32con.WS_VISIBLE         |
        win32con.WS_CAPTION     |
        win32con.WS_SYSMENU     |
        win32con.DS_SETFONT)
di = ['Python',                                 # 设置对话框属性，设置标题为 "Python"
    (0,0,300,180),                              # 设置对话框位置及大小
    style,                                      # 设置对话框样式
    None,                                       # 设置对话框扩展样式
    (8, "MS Sans Serif")]                       # 设置对话框字体及大小
init = []                                       # 定义对话框初始化信息列表
init.append(di)
mydialog = MyDialog(init)                       # 生成对话框实例对象
mydialog.DoModal()                              # 显示对话框
```

运行 SimpleDialog.py 脚本后，将创建如图 11-3 所示一个简单的对话框。

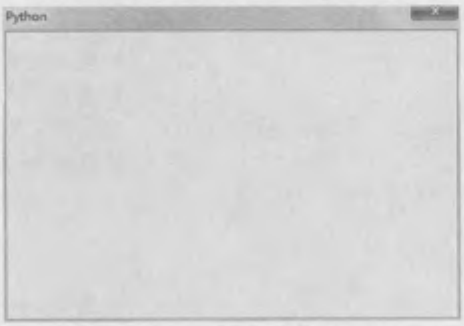


图 11-3 一个简单的对话框

11.2.2 向对话框添加控件

在 11.2.1 节的例子中，只创建了一个简单的只有外边框的对话框，而一个较完整的对话框通常应包含按钮、文本框等控件，以方便用户操作。

使用 pywin.mfc 模块中的 Dialog 模块创建对话框时，可以将控件信息放在初始化参数中。当创建对话框时将创建相应的控件。

使用 PythonWin 创建控件时，需要使用一个列表来保存控件的信息，列表由以下几项组成。

- 控件类型，可以为字符串或者数字，如文本框、按钮等。
- 控件标题。
- 控件 ID。
- 控件的位置。为包含 4 项的元组，其分别为距对话框左上角的 *X* 坐标、*Y* 坐标、宽度以及高度。
- 控件样式。

其中，常用控件类型如表 11-1 所示。

表 11-1 控件类型

控件名称	字符表示	数字表示
按钮	Button	128
组合框	ComboBox	133
文本框	Edit	129
列表框	ListBox	131
滚动条	ScrollBar	132
标签	Static	130

下面所示的 Dialog.py 脚本创建了一个对话框，并向对话框中添加了文本框、按钮、标签等控件。

```
# -*- coding:utf-8 -*-
# file: Dialog.py
#
import win32ui                                  # 导入 win32ui 模块
import win32con                                 # 导入 win32con 模块
from pywin.mfc import dialog                    # 从 pywin.mfc 导入 dialog
class MyDialog(dialog.Dialog):                  # 通过继承 dialog.Dialog 生成对话框类
    def OnInitDialog(self):                     # 重载对话框初始化方法
        dialog.Dialog.OnInitDialog(self)        # 调用父类的对话框初始化方法
    def OnOK(self):                             # 重载 OnOK 方法
        win32ui.MessageBox('Press Ok', \
                'Python',                \
                win32con.MB_OK)
        self.EndDialog(1)
    def OnCancel(self):                         # 重载 OnCancel 方法
        win32ui.MessageBox('Press Cancel',\
                'Python',     \
                win32con.MB_OK)
```

```
        self.EndDialog(1)
style = (win32con.DS_MODALFRAME |                    # 定义对话框样式
        win32con.WS_POPUP   |
        win32con.WS_VISIBLE |
        win32con.WS_CAPTION |
        win32con.WS_SYSMENU |
        win32con.DS_SETFONT)
childstyle = (win32con.WS_CHILD |                    # 定义控件样式
     win32con.WS_VISIBLE)
buttonstyle = win32con.WS_TABSTOP | childstyle # 定义按钮样式
di = ['Python',                                      # 设置对话框属性
    (0,0,300,180),
    style,
    None,
    (8, "MS Sans Serif")]
ButOK = (['Button',                                  # 设置 OK 按钮属性
    "OK",
    win32con.IDOK,
    (80,150, 50, 14),
    buttonstyle | win32con.BS_PUSHBUTTON])
ButCancel = (['Button',                              # 设置 Cancel 按钮属性
        "Cancel",
        win32con.IDCANCEL,
        (160, 150, 50, 14),
        buttonstyle | win32con.BS_PUSHBUTTON])
Stadic = (['Static',                                 # 设置标签属性
        'Python Dialog',
        12,
        (130,50,60,14),
        childstyle])
Edit = (['Edit',                                     # 设置文本框属性
        '',
        13,
        (130,80,60,14),
        childstyle|win32con.ES_LEFT|
        win32con.WS_BORDER|win32con.WS_TABSTOP])
init = []                                            # 初始化信息列表
init.append(di)
init.append(ButOK)
init.append(ButCancel)
init.append(Stadic)
init.append(Edit)
mydialog = MyDialog(init)                            # 生成对话框实例对象
mydialog.DoModal()                                   # 创建对话框
```

运行 Dialog.py 脚本后将创建如图 11-4 所示的对话框。



图 11-4 添加了控件的对话框

11.2.3 使用 DLL 文件中的资源

从前面编写的代码可以看出，在 Python 中直接定义对话框中的控件比较烦琐。实际上，win32ui 模块中的 LoadDialogResource 函数可以载入 DLL 文件中的对话框资源。这样只要在 VS2008 或者其他 VC 开发环境中创建一个 DLL 工程，在这个工程中用 C++创建出需要添加的对话框及控件，然后就可以在 Python 中通过 win32ui.LoadDialogResource 函数载入这些资源，直接使用其中的对话框及控件等。

在使用 DLL 中的资源之前，应使用 win32ui.LoadLibrary 函数载入 DLL 文件。其函数原型如下。

```
LoadLibrary(fileName)
```

其参数含义如下。

- filename　　要载入的 DLL 文件名。

当 win32ui.LoadLibrary 函数正确载入 DLL 文件后，将返回一个 PyDLL 类型的值，然后就可以通过该值及对话框的 ID 来使用 win32ui.LoadDialogResource 函数载入 DLL 中的对话框了。

win32ui.LoadDialogResource 函数原型如下所示。

```
LoadDialogResource(idRes, dll )
```

其参数含义如下。

- idRes　　对话框的 ID。
- dll　　使用 win32ui.LoadLibrary 载入 DLL 文件的返回值。

win32ui.LoadDialogResource 函数会返回一个资源列表，在对话框类的实例化时调用该列表即可使用 DLL 文件中的对话框。

其操作步骤如下。

首先在 VS2008 中创建一个名为“Res”的“MFC AppWizard(dll)”工程，按照默认选项设置工程。然后向工程中添加对话框及相应的资源，完成后编译 DLL。最后，打开工程中的“Resource.h”文件，查看对话框的 ID。在“Resource.h”会找到类似下面所示的语句。

```
#define IDD_DIALOG1                     7000
```

其中，7000 是对话框的 ID，即 LoadDialogResource 函数中的 idRes 参数。下面所示的 DllRes.py 脚本通过使用“Res.dll”文件中的资源创建对话框。

```
# -*- coding:utf-8 -*-
# file: DllRes.py
#
import win32ui                                  # 导入 win32ui 模块
import win32con                                 # 导入 win32con 模块
from pywin.mfc import dialog                    # 导入 pywin.mfc 中的 dialog 模块
class MyDialog(dialog.Dialog):                  # 通过继承 dialog.Dialog 生成对话框类
    def OnInitDialog(self):                     # 重载初始化函数
        dialog.Dialog.OnInitDialog(self)        # 调用父类的初始化函数
```

```
    def OnOK(self):                                    # 重载 OnOK 方法
        win32ui.MessageBox('Press Ok', \
                'Python',                \
                win32con.MB_OK)
        self.EndDialog(1)
    def OnCancel(self):                                # 重载 OnCancel 方法
        win32ui.MessageBox('Press Cancel',\
                'Python',                \
                win32con.MB_OK)
        self.EndDialog(1)
dll = win32ui.LoadLibrary('Res.dll')                   # 载入 DLL 文件
l = win32ui.LoadDialogResource(7000,dll)               # 载入 DLL 文件中的对话框
mydialog = MyDialog(l)                                 # 生成对话框实例对象
mydialog.DoModal()                                     # 显示对话框
```

运行 DllRes.py 脚本后，将创建如图 11-5 所示的对话框。



图 11-5 使用 DLL 中的资源创建对话框

11.2.4 处理按钮消息

在前面的例子中，是通过重载了父类的 OnOK 方法来处理 OK 按钮单击的消息。如果添加其他的按钮，则需要使用 HookCommand 方法为按钮设置相应的消息处理方法，其原型如下。

```
HookCommand(method, id)
```

其参数含义如下。

- method　　处理消息的方法。
- id　　按钮的 ID。

其中，按钮消息处理的方法应定义为下面的形式。

```
def OnButton(self,lParam,wParam):                      # 处理 Button 单击消息
    <消息处理语句>
```

在上面的函数中，除了必须的 self 参数外，还应该有 wParam 和 lParam 参数。其中 wParam 参数的值总是为 0，而 lParam 参数的值则为所单击按钮的 ID。下面所示的 DialogCmd.py 脚本是使用 HookCommand 方法来处理按钮消息。

```
# -*- coding:utf-8 -*-
# file: DialogCmd.py
#
```

```
import win32ui                                          # 导入 win32ui 模块
import win32con                                         # 导入 win32con 模块
from pywin.mfc import dialog                            # 从 pywin.mfc 导入 dialog
class MyDialog(dialog.Dialog):                          # 通过继承 dialog.Dialog 生成对话框类
    def OnInitDialog(self):                             # 重载对话框初始化方法
        dialog.Dialog.OnInitDialog(self)                # 调用父类的对话框初始化方法
        self.HookCommand(self.OnButton1,1051)           # 设置按钮消息处理方法
        self.HookCommand(self.OnButton2,1052)
    def OnButton1(self,wParam,lParam):                  # 处理 Button1 单击消息
        win32ui.MessageBox('Button1',   \
                'Python',   \
                win32con.MB_OK)
        self.EndDialog(1)
    def OnButton2(self,wParam,lParam):                  # 处理 Button2 单击消息
        text = self.GetDlgItemText(1054)                # 获得文本框中的文字
        win32ui.MessageBox(text,    \
                'Python',   \
                win32con.MB_OK)
        self.EndDialog(1)
style = (win32con.DS_MODALFRAME |                       # 定义对话框样式
        win32con.WS_POPUP   |
        win32con.WS_VISIBLE |
        win32con.WS_CAPTION |
        win32con.WS_SYSMENU |
        win32con.DS_SETFONT)
childstyle = (win32con.WS_CHILD |                       # 定义控件样式
     win32con.WS_VISIBLE)
buttonstyle = win32con.WS_TABSTOP | childstyle          # 定义按钮样式
di = ['Python',                                         # 设置对话框属性
    (0,0,300,180),
    style,
    None,
    (8, "MS Sans Serif")]
Button1 = (['Button',                                   # 设置 Button1 按钮属性
    'Button1',
    1051,
    (80,150, 50, 14),
    buttonstyle | win32con.BS_PUSHBUTTON])
Button2 = (['Button',                                   # 设置 Button2 按钮属性
    'Button2',
    1052,
    (160, 150, 50, 14),
    buttonstyle | win32con.BS_PUSHBUTTON])
Stadic = (['Static',                                    # 设置标签属性
    'Python Dialog',
    1053,
    (130,50,60,14),
    childstyle])
Edit = (['Edit',                                        # 设置文本框属性
    '',
    1054,
    (130,80,60,14),
    childstyle|win32con.ES_LEFT|
    win32con.WS_BORDER|win32con.WS_TABSTOP])
```

```
init = []                                          # 初始化信息列表
init.append(di)
init.append(Button1)
init.append(Button2)
init.append(Stadic)
init.append(Edit)
mydialog = MyDialog(init)                          # 生成对话框实例对象
mydialog.DoModal()                                 # 创建对话框
```

运行 DialogCmd.py 脚本后，单击 Button1 按钮，对话框如图 11-6 所示。在文本框中输入“Python!”后，单击 Button2 按钮，对话框如图 11-7 所示。

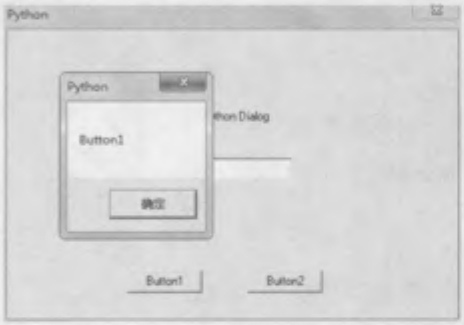


图 11-6 单击 Button1 按钮



图 11-7 单击 Button2 按钮

11.3 创建菜单

菜单是 GUI 程序中重要的组成部分，通过菜单用户可以使用程序提供的各项功能，如打开文件、关闭文件等。对于较为大型的软件，由于操作功能较多，因此通常有很多菜单项来完成各种功能。

11.3.1 创建菜单

通过使用 PythonWin 相关模块提供的函数，可以在脚本中动态地创建菜单，然后将创建的菜单添加到窗口中。

1. 创建普通菜单

使用 win32ui 模块中的 CreateMenu 可以动态地创建菜单，其返回值为 PyCMenu 对象。使用 PyCMenu 对象的 AppendMenu 方法可以向其中添加菜单。其原型如下。

```
AppendMenu(flags, id, value)
```

其参数含义如下。

- flags　　菜单样式。
- id　　菜单的 ID。
- value　　菜单名称。

其中，参数 flags 应为 win32con.MF_POPUP、win32con.MF_SEPARATOR、win32con.MF_STRING、

win32con.MF_UNCHECKED 等。value 可以为 None，但如果 value 为字符串，则 flags 中必须包含 win32con.MF_STRING 标志。

下面所示的 CreateMenu.py 脚本是使用 CreateMenu 创建几种菜单。

```
# -*- coding:utf-8 -*-
# file: UseMenu.py
#
import win32ui
import win32api
from win32con import *
from pywin.mfc import window
# 定义窗口类
class MyWnd(window.Wnd):
    def __init__ (self):
        window.Wnd.__init__(self, win32ui.CreateWnd ())
        self._obj_.CreateWindowEx(WS_EX_CLIENTEDGE, \
                win32ui.RegisterWndClass(0, 0, COLOR_WINDOW + 1), \
                'MFC GUI', WS_OVERLAPPEDWINDOW, \
                (10, 10, 800, 500), None, 0, None)
        # 创建菜单对象
        submenu = win32ui.CreateMenu()
        menu = win32ui.CreateMenu()
        # 向菜单中添加 Open，其中，&表示可以使用 Alt+&后的字母访问该菜单命令
        submenu.AppendMenu(MF_STRING,1051,'&Open')
        # 向菜单中添加 Close
        submenu.AppendMenu(MF_STRING,1052,'&Close')
        # 向菜单中添加 Save
        submenu.AppendMenu(MF_STRING,1053,'&Save')
        # 将上面的菜单添加到 File 菜单中
        menu.AppendMenu(MF_STRING|MF_POPUP,submenu.GetHandle(),'&File')
        submenu = win32ui.CreateMenu()
        # 向菜单中添加 Copy
        submenu.AppendMenu(MF_STRING,1054,'&Copy')
        # 向菜单中添加 Paste
        submenu.AppendMenu(MF_STRING,1055,'&Paste')
        # 向菜单中添加分隔符
        submenu.AppendMenu(MF_SEPARATOR,1056,None)
        # 向菜单中添加 Cut
        submenu.AppendMenu(MF_STRING,1057,'C&ut')
        # 将上面的菜单添加到 Edit 菜单中
        menu.AppendMenu(MF_STRING|MF_POPUP,submenu.GetHandle(),'&Edit')
        submenu = win32ui.CreateMenu()
        # 向菜单中添加 Tools
        submenu.AppendMenu(MF_STRING,1058,'Tools')
        # 向菜单中添加 Settings
        submenu.AppendMenu(MF_STRING|MF_GRAYED,1059,'Setting')
        m = win32ui.CreateMenu()
        # 将上面的菜单添加到 Option 菜单中
        m.AppendMenu(MF_STRING|MF_POPUP|MF_CHECKED,submenu.GetHandle(),'Option')
        # 将 Option 菜单添加到 Other 菜单中
        menu.AppendMenu(MF_STRING|MF_POPUP,m.GetHandle(),'&Other')
        # 将菜单添加到窗口中
        self._obj_.SetMenu(menu)
```

```
    # 重载 OnClose 方法
    def OnClose(self):
        self.EndModalLoop(0)
w = MyWnd()                                    # 生成窗口对象
w.ShowWindow()                                 # 显示窗口
w.UpdateWindow()                               # 刷新窗口
w.RunModalLoop(1)                              # 进入消息循环
```

运行脚本 CreateMenu.py 后，将创建如图 11-8、图 11-9 和图 11-10 所示的菜单。



图 11-8 File 菜单



图 11-9 Edit 菜单



图 11-10 Other 菜单

2. 创建弹出式菜单

使用 win32ui 模块中的 CreatePopupMenu，可以动态地创建弹出式菜单，其返回值同 CreateMenu 一样，也是 PyCMenu 对象。在使用 CreatePopupMenu 创建完菜单后，即可使用 PyCMenu 对象的 AppendMenu 方法向其中添加菜单。下面所示的 PopupMenu.py 脚本创建了一个简单的窗口，当右击窗口时将弹出一个菜单。

```
# -*- coding:utf-8 -*-
# file: PopupMenu.py
#
import win32ui
import win32api
from win32con import *
from pywin.mfc import window
# 定义窗口类
class MyWnd(window.Wnd):
    def __init__ (self):
        window.Wnd.__init__(self, win32ui.CreateWnd ())
        self._obj_.CreateWindowEx(WS_EX_CLIENTEDGE, \
```

```
                win32ui.RegisterWndClass(0, 0, COLOR_WINDOW + 1), \
                'MFC GUI', WS_OVERLAPPEDWINDOW, \
                (10, 10, 800, 500), None, 0, None)
        # 捕获右键单击消息
        self.HookMessage(self.OnRClick,WM_RBUTTONDOWN)
    # 重载 OnClose 方法
    def OnClose(self):
        self.EndModalLoop(0)
    # 处理单击鼠标右键消息
    def OnRClick(self, param):
        submenu = win32ui.CreatePopupMenu()
        submenu.AppendMenu(MF_STRING,1054,'Copy')          # 向菜单中添加 Copy
        submenu.AppendMenu(MF_STRING,1055,'Paste')         # 向菜单中添加 Paste
        submenu.AppendMenu(MF_SEPARATOR,1056,None)         # 向菜单中添加分隔符
        submenu.AppendMenu(MF_STRING,1057,'Cut')           # 向菜单中添加 Cut
        flag = TPM_LEFTALIGN|TPM_LEFTBUTTON|TPM_RIGHTBUTTON    # 弹出式菜单样式
        # param 为系统向 OnRClick 函数传递的参数，其为一个元组，其第 6 项为鼠标 X，Y 坐标组成
的元组
        submenu.TrackPopupMenu(param[5],flag,self)
w = MyWnd()                                                # 生成窗口对象
w.ShowWindow()                                             # 显示窗口
w.UpdateWindow()                                           # 刷新窗口
w.RunModalLoop(1)                                          # 进入消息循环
```

运行脚本 PopupMenu.py 后，在创建的窗口中单击右键将弹出如图 11-11 所示的菜单。



图 11-11　弹出式菜单

11.3.2　使用 DLL 中的菜单

在 11.3.1 节中演示的创建菜单的方法都是在 Python 脚本中直接创建，如果窗口中菜单较多，那么创建菜单的代码就会很多，而且创建过程十分烦琐。与使用 DLL 文件中的对话框资源相同，也可以在 Python 中直接载入 DLL 文件中的菜单。

使用 win32ui.LoadMenu 函数可以载入 DLL 文件中的菜单。其函数原型如下。

```
LoadMenu(id, dll)
```

其参数含义如下。

- ◆ idRes　　菜单的 ID。
- ◆ dll　　使用 win32ui.LoadLibrary 载入 DLL 文件的返回值。

可以直接使用 11.3.1 节中创建的“Res”工程，向其中添加菜单。编译完成后，可打开工程中的“Resource.h”文件查看菜单的 ID。下面所示的 DllMenu.py 脚本可载入 DLL 文件中的菜单，并将其添加到窗口中。

```
# -*- coding:utf-8 -*-
# file: DllMenu.py
#
import win32ui
import win32api
from win32con import *
from pywin.mfc import window
# 定义窗口类
class MyWnd(window.Wnd):
    def __init__ (self):
        window.Wnd.__init__(self, win32ui.CreateWnd ())
        self._obj_.CreateWindowEx(WS_EX_CLIENTEDGE, \
                win32ui.RegisterWndClass(0, 0, COLOR_WINDOW + 1), \
                'MFC GUI', WS_OVERLAPPEDWINDOW, \
                (10, 10, 800, 500), None, 0, None)
        dll = win32ui.LoadLibrary('Res.dll')                    # 载入 DLL 文件
        m = win32ui.LoadMenu(7001,dll)                          # 载入菜单
        self._obj_.SetMenu(m)                                   # 向创建添加菜单
    # 重载 OnClose 方法
    def OnClose(self):
        self.EndModalLoop(0)
    # 重载 OnPaint 方法，在窗口中输出 "MFC GUI"
    def OnPaint(self):
        dc,ps = self.BeginPaint()
        dc.DrawText('MFC GUI',
            self.GetClientRect(),
            DT_SINGLELINE | DT_CENTER | DT_VCENTER)
        self.EndPaint(ps)

w = MyWnd()                                                     # 生成窗口对象
w.ShowWindow()                                                  # 显示窗口
w.UpdateWindow()                                                # 刷新窗口
w.RunModalLoop(1)                                               # 进入消息循环
```

运行脚本 DllMenu.py 后，将创建如图 11-12、图 11-13 和图 11-14 所示的菜单。



图 11-12 File 菜单



图 11-13 Edit 菜单

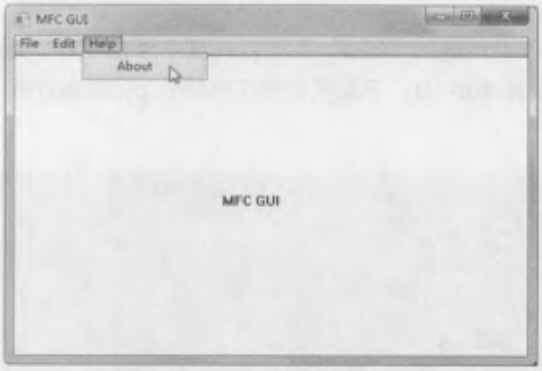


图 11-14 Help 菜单

11.3.3 处理菜单消息

菜单消息的处理与按钮消息的处理过程相同，只需使用 HookCommand 方法设置相应菜单的消息处理方法即可。下面所示的 MenuCmd.py 脚本是使用 HookCommand 方法处理菜单的消息。

```
# -*- coding:utf-8 -*-
# file: MenuCmd.py
#
import win32ui
import win32api
from win32con import *
from pywin.mfc import window
# 定义窗口类
class MyWnd(window.Wnd):
    def __init__ (self):
        window.Wnd.__init__(self, win32ui.CreateWnd ())
        self._obj_.CreateWindowEx(WS_EX_CLIENTEDGE, \
                win32ui.RegisterWndClass(0, 0, COLOR_WINDOW + 1), \
                'MFC GUI', WS_OVERLAPPEDWINDOW, \
                (10, 10, 800, 500), None, 0, None)
        # 创建菜单对象
        submenu = win32ui.CreateMenu()
        menu = win32ui.CreateMenu()
        # 向菜单中添加 Open，其中&表示可以使用 Alt+&后的字母访问该菜单命令
        submenu.AppendMenu(MF_STRING,1051,'&Open')
        # 向菜单中添加 Close
        submenu.AppendMenu(MF_STRING,1052,'&Close')
        # 向菜单中添加 Save
        submenu.AppendMenu(MF_STRING,1053,'&Save')
        # 将上面的菜单添加到 File 菜单中
        menu.AppendMenu(MF_STRING|MF_POPUP,submenu.GetHandle(),'&File')
        # 将菜单添加到窗口中
        self._obj_.SetMenu(menu)
        # 设置菜单处理消息
        self.HookCommand(self.MenuClick,1051)
        self.HookCommand(self.MenuClick,1052)
        self.HookCommand(self.MenuClick,1053)
        # 重载 OnClose 方法
    def OnClose(self):
```

```
        self.EndModalLoop(0)
    def MenuClick(self,lParam,wParam):
        # 根据 lParam 参数判断单击的菜单
        if lParam == 1051:
            self.MessageBox('Open','Python',MB_OK)
        elif lParam == 1053:
            self.MessageBox('Save','Python',MB_OK)
        else:
            self.OnClose()
w = MyWnd()                                          # 生成窗口对象
w.ShowWindow()                                       # 显示窗口
w.UpdateWindow()                                     # 刷新窗口
w.RunModalLoop(1)                                    # 进入消息循环
```

运行 MenuCmd.py 脚本，单击菜单【File】|【Open】命令，如图 11-15 所示。单击菜单【File】|【Save】命令，如图 11-16 所示。单击菜单【File】|【Close】命令，将关闭窗口。



图 11-15 单击【Open】菜单命令



图 11-16 单击【Save】菜单命令

11.4 本章小结

本章介绍了用 Python 编写图形用户界面（GUI）的一种方法，即调用 Windows GUI 来创建图形用户界面。首先介绍了使用 PythonWin 中的 win32gui 和 win32ui 模块创建 Windows API 及 MFC 创建窗口的方法，接着介绍了创建对话框、菜单、处理菜单消息等常规的 GUI 设计方法。通过对本章的学习，我们可以用 Python 创建一个 Windows 类型的应用程序。

本章的方法只局限于在 Windows 环境下使用，而下一章将介绍使用 tkinter 设计 GUI，这是一种可在多个操作系统中使用的 GUI 模块。

◆ ◆ ◆

第 12 章　使用 tkinter 编写 GUI

本章包括

- tkinter 概述
- 处理 tkinter 组件的事件
- 创建自定义对话框
- 使用 tkinter 组件
- 使用 tkinter 标准对话框

tkinter 是 Python 自带的用于 GUI 编程的模块，是对图形库 TK 的封装。tkinter 是跨平台的，这意味着在 Windows 下编写的脚本，可以不加修改地在 Linux、UNIX 等系统下运行。而第 11 章中用 MFC 编写的 GUI 程序则仅适用于 Windows 平台，不能在 Linux、UNIX 等平台大运行。因此，tkinter 的优势在于其可移植性。

12.1　tkinter 概述

使用 tkinter 可以创建完整的 GUI 程序。在 tkinter 中，文本框、按钮、标签等在第 11 章中所提到的控件都被称之为组件（widget），也就是说，tkinter 中的组件相当于第 11 章中所提到的控件。

tkinter 是 Python 的一个模块，可以像其他模块一样在 Python 的交互式 shell 中（或者“.py”脚本中）被导入，tkinter 模块被导入后，即可使用 tkinter 模块中的函数、方法等。

12.1.1　创建简单的窗口

使用 tkinter 创建图形界面时首先要导入 tkinter 模块。可以在 Python 的交互式 shell 中输入以下语句来验证 Python 是否安装了 tkinter 模块。

```
import tkinter
```

如果上述语句执行成功，则表示已经安装了 tkinter 模块。在编写脚本时，只要使用 import 语句导入了 tkinter 模块，即可使用 tkinter 模块中的函数、对象等进行 GUI 编程。

在使用 tkinter 模块时，首先要使用 tkinter.Tk 生成一个主窗口对象，然后才能使用 tkinter 模块中其他的函数、方法等。当生成主窗口后，可以向其添加组件，或者直接调用其 mainloop 方法进行消息循环。下面所示的 tkinterWindow.py 脚本仅创建了一个简单的窗口，而没有使用组件。

```
# -*- coding:utf-8 -*-
# file: tkinterWindow.py
#
import tkinter                                  # 导入 tkinter 模块
root = tkinter.Tk()                             # 生成 root 主窗口
root.mainloop()                                 # 进入消息循环
```

运行脚本 tkinterWindow.py 后，将创建如图 12-1 所示的窗口。



图 12-1 简单的 tkinter 窗口

上述脚本与第 11 章中使用 MFC 创建窗口的脚本相比更为简单，仅使用 3 条语句就创建了一个简单的 GUI 窗口。当完成窗口内部组件的创建工作后，也要进入消息循环中，以处理窗口及其内部组件的事件。

如果需要在 tkinter 的窗口、组件中显示中文，除了需要在 ".py" 脚本文件中的首行添加 "# -*- coding:utf-8 -*-" 指明字符编码外，还应将脚本保存成 "UTF-8" 的编码格式。如果使用记事本，则可以单击【文件】|【另存为】命令，在【另存为】对话框下方的【编码】下拉框中，选择 "UTF-8" 选项。然后单击【保存】按钮，即可在 tkinter 的窗口或组件中使用中文。

12.1.2 向窗口中添加组件

在 tkinter 中，组件是由 tkinter 模块中相应的组件函数生成。组成生成后就可以使用 pack 方法、grid 方法，或者 place 方法将其添加到窗口中。下面所示的 Hellotkinter.py 脚本是对 12.1.1 节中的例子改写，向窗口中添加了标签和按钮组件。

```
# -*- coding:utf-8 -*-
# file: Hellotkinter.py
#
import tkinter                                          # 导入 tkinter 模块
root = tkinter.Tk()                                     # 生成 root 主窗口
label= tkinter.Label(root, text="Hello, tkinter!") # 生成标签
label.pack()                                            # 将标签添加到 root 主窗口
button1 = tkinter.Button(root, text="Button1")      # 生成 button1
button1.pack(side=tkinter.LEFT)                         # 将 button1 添加到 root 主窗口
button2 = tkinter.Button(root, text="Button2")      # 生成 button2
button2.pack(side=tkinter.RIGHT)                        # 将 button2 添加到 root 主窗口
root.mainloop()                                         # 进入消息循环
```

运行 Hellotkinter.py 脚本后将创建如图 12-2 所示的窗口。



图 12-2 向窗口中添加组件

在使用 tkinter 向窗口中添加组件时，如果不指定组件的位置信息，则 tkinter 自动将组件安排在合适的位置。当然，组件的位置也可以精确控制，达到所需的效果。

在上述脚本中，使用了标签和按钮组件。除此之外，tkinter 还提供了其他的组件，以适合不同的需要。

运行 Hellotkinter.py 脚本后，单击两个按钮均无反映，这是因为在脚本中未添加按钮组件单击事件的处理函数。关于组件的事件处理将在本章第 3 节中进行讲解，此处仅给出一个简单的创建窗口的例子。

12.2 使用组件

12.1 节中创建的窗口实际上是存放组件的一个“容器”。如果仅创建一个不包含组件的窗口，则其作用也仅是测试 tkinter 模块。更有意义的做法是，当窗口创建好以后，应根据脚本的功能向窗口中添加合适的组件，然后定义相关实际的处理函数，这样才算一个完整的 GUI 程序。

12.2.1 组件分类

tkinter 中包含了 15 种核心组件，以实现不同的功能，如表 12-1 所示。

表 12-1 tkinter 中的组件

组件名称	描述
Button	按钮
Canvas	绘制图形组件，可以在其中绘制图形
Checkbutton	复选框
Entry	文本框（单行）
Frame	框架，将几个组件组成一组
Label	标签，可以显示文字或者图片
Listbox	列表框
Menu	菜单
Menubutton	Menubutton 的功能完全可以使用 Menu 替代
Message	与 Label 组件类似，但是可以根据自身大小将文本换行
Radiobutton	单选框
Scale	滑块
Scrollbar	滚动条
Text	文本框（多行）
Toplevel	用来创建子窗口容器组件

12.2.2 组件布局

在 12.1.2 节的例子中，仅仅使用组件的 pack 方法将组件添加到窗口中，而未设置组件的位置，因此，组件位置都是由 tkinter 模块自动确定的。

对于包含较多组件的窗口，为了让组件布局合理，可以通过向 pack 传递参数来设置组件在窗口中的位置。除了组件的 pack 方法外，还可以使用 grid 方法和 place 方法来放置组件。

组件的 pack 方法可以使用以下几个参数来设置组件的位置属性。

- after　　将组件置于其他组件之后。
- anchor　　组件的对齐方式，顶对齐“n”、底对齐“s”、左对齐“w”、右对齐“e”。

- before　　将组件置于其他组件之前。
- side　　组件在主窗口的位置，可以为“top”、“bottom”、“left”、“right”。

组件的 grid 方法是使用行列的方法来放置组件的位置，其主要使用以下几个参数设置组件的位置属性。

- column　　组件所在的列起始位置。
- columnspam　　组件的列宽。
- row　　组件所在的行起始位置。
- rowspam　　组件的行宽。

组件的 place 方法相对较为灵活，可以直接使用坐标来放置组件的位置，其主要使用以下几个参数设置组件的位置属性。

- anchor　　组件对齐方式。
- x　　组件左上角的 X 坐标。
- y　　组件左上角的 Y 坐标。
- relx　　组件相对于窗口的 X 坐标，应为 0～1 之间的小数。
- rely　　组件相对于窗口的 Y 坐标，应为 0～1 之间的小数。
- width　　组件的宽度。
- height　　组件的高度。
- relwidth　　组件相对于窗口的宽度，应为 0～1 之间的小数。
- relheight　　组件相对于窗口的高度，应为 0～1 之间的小数。

12.2.3 使用按钮

使用 tkinter.Button 时，向其传递参数可以控制按钮的属性，例如，可以设置按钮上文本的颜色、按钮的颜色、按钮的大小以及按钮的状态等。常用的控制参数如下。

- anchor　　指定按钮上文本的位置。
- background（bg）　　指定按钮的背景色。
- bitmap　　指定按钮上显示的位图。
- borderwidth（bd）　　指定按钮边框的宽度。
- command　　指定按钮消息的回调函数。
- cursor　　指定鼠标移动到按钮上的指针样式。
- font　　指定按钮上文本的字体。
- foreground（fg）　　指定按钮的前景色。
- height　　指定按钮的高度。
- image　　指定按钮上显示的图片。
- state　　指定按钮的状态。
- text　　指定按钮上显示的文本。
- width　　指定按钮的宽度。

下面所示的 tkinterButton.py 脚本创建了各种不同的按钮。

```
# -*- coding:utf-8 -*-
# file: tkinterButton.py
#
import tkinter                                  # 导入 tkinter 模块
root = tkinter.Tk()
button1 = tkinter.Button(root,
            anchor = tkinter.E,                 # 指定文本对齐方式
            text = 'Button1',                   # 指定按钮上的文本
            width = 40,                         # 指定按钮的宽度，相当于 40 个字符
            height = 5)                         # 指定按钮的高度，相当于 5 行字符
button1.pack()                                  # 将按钮添加到窗口
button2 = tkinter.Button(root,
            text = 'Button2',
            bg = 'blue')                        # 指定按钮的背景色
button2.pack()
button3 = tkinter.Button(root,
            text = 'Button3',
            width = 14,                         # 指定按钮的宽度
            height = 1)                         # 指定按钮的高度
button3.pack()
button4 = tkinter.Button(root,
            text = 'Button4',
            width = 60,
            height = 5,
            state = tkinter.DISABLED)           # 指定按钮为禁用状态
button4.pack()
root.mainloop()                                 # 进入消息循环
```

运行 tkinterButton.py 脚本后将创建如图 12-3 所示的 4 个按钮。

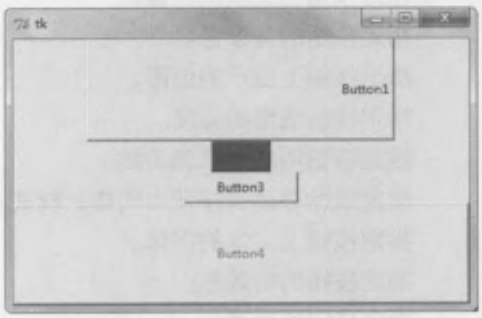


图 12-3　创建的不同类型的按钮

12.2.4　使用文本框

文本框主要用来接收用户输入。

使用 tkinter.Entry 和 tkinter.Text 可以创建单行文本框和多行文本框组件。通过向其传递参数可以设置文本框的背景色、大小、状态等。以下是 tkinter.Entry 和 tkinter.Text 共有的几个控制参数。

- background（bg）　指定文本框的背景色。
- borderwidth（bd）　指定文本框边框的宽度。
- font　指定文本框中文字的字体。
- foreground（fg）　指定文本框的前景色。
- selectbackground　指定选定文本的背景色。
- selectforeground　指定选定文本的前景色。
- show　指定文本框中显示的字符，若为 "*"，则表示文本框为密码框。
- state　指定文本框的状态。
- width　指定文本框的宽度。

下面所示的 tkinterEntry.py 脚本创建了各种不同类型的文本框。

```
# -*- coding:utf-8 -*-
# file: tkinterEntry.py
#
import tkinter                                  # 导入 tkinter 模块
root = tkinter.Tk()
entry1 = tkinter.Entry(root,                    # 生成单行文本框组件
           show = '*',)                         # 输入文本框中的字符被显示为 "*"
entry1.pack()                                   # 将文本框添加到窗口中
entry2 = tkinter.Entry(root,
           show = '#',                          # 输入文本框中的字符被显示为 "#"
           width = 50)                          # 将文本框的宽度设置为 50
entry2.pack()
entry3 = tkinter.Entry(root,
           bg = 'red',                          # 将文本框的背景色设置为红色
           fg = 'blue')                         # 将文本框的前景色设置为蓝色
entry3.pack()
entry4 = tkinter.Entry(root,
           selectbackground = 'red',            # 将选中文本的背景色设置为红色
           selectforeground = 'gray')           # 将选中文本的前景色设置为灰色
entry4.pack()
entry5 = tkinter.Entry(root,
           state = tkinter.DISABLED)            # 将文本框设置为禁用
entry5.pack()
edit1 = tkinter.Text(root,                      # 生成多行文本框组件
           selectbackground = 'red',            # 将选中文本的背景色设置为红色
           selectforeground = 'gray')           # 将选中文本的前景色设置为灰色
edit1.pack()
root.mainloop()                                 # 进入消息循环
```

运行 tkinterEntry.py 脚本后，将创建如图 12-4 所示的 6 个文本框。



图 12-4　创建的不同类型的文本框

12.2.5 使用标签

标签是供在窗口中显示文本的组件。除了显示文本外，标签还可以显示图片。使用 tkinter.Label 可以创建标签组件。以下是其几个常用的属性，通过这些属性可控制标签。

- anchor　　指定标签中文本的位置。
- background（bg）　　指定标签的背景色。
- borderwidth（bd）　　指定标签的边框宽度。
- bitmap　　指定标签中的位图。
- font　　指定标签中文本的字体。
- foreground（fg）　　指定标签的前景色。
- height　　指定标签的高度。
- image　　指定标签中的图片。
- justify　　指定标签中多行文本的对齐方式。
- text　　指定标签中的文本，可以使用“\n”表示换行。
- width　　指定标签的宽度。

下面所示的 tkinterLabel.py 脚本创建了几种不同类型的标签组件。

```
# -*- coding:utf-8 -*-
# file: tkinterLabel.py
#
import tkinter                                  # 导入 tkinter 模块
root = tkinter.Tk()
label1 = tkinter.Label(root,
            anchor = tkinter.E,                 # 设置文本的位置
            bg = 'blue',                        # 设置标签背景色
            fg = 'red',                         # 设置标签前景色
            text = 'Python',                    # 设置标签中的文本
            width = 30,                         # 设置标签的宽度为 30
```

```
            height = 5)                          # 设置标签的高度为 5
label1.pack()
label2 = tkinter.Label(root,
            text = 'Python GUI\ntkinter',        # 设置标签中的文本，在字符串中使用换行符
            justify = tkinter.LEFT,              # 设置多行文本为左对齐
            width = 30,
            height = 5)
label2.pack()
label3 = tkinter.Label(root,
            text = 'Python GUI\ntkinter',
            justify = tkinter.RIGHT,             # 设置多行文本为右对齐
            width = 30,
            height = 5)
label3.pack()
label4 = tkinter.Label(root,
            text = 'Python GUI\ntkinter',
            justify = tkinter.CENTER,            # 设置多行文本为居中对齐
            width = 30,
            height = 5)
label4.pack()
root.mainloop()
```

运行 tkinterLabel.py 脚本后将创建如图 12-5 所示的 4 个不同类型的标签组件。



图 12-5 不同类型的标签

12.2.6 使用菜单

在 tkinter 中，菜单组件的添加与其他组件有所不同。菜单需要使用所创建的主窗口的 config 方法添加到窗口中。下面所示的 tkinterMenu.py 脚本向主窗口中添加了菜单。

```
# -*- coding:utf-8 -*-
# file: tkinterMenu.py
#
import tkinter                                   # 导入 tkinter 模块
root = tkinter.Tk()
menu = tkinter.Menu(root)                        # 生成菜单
submenu = tkinter.Menu(menu, tearoff=0)          # 生成下拉菜单
submenu.add_command(label="Open")                # 向下拉菜单中添加 Open 命令
submenu.add_command(label="Save")                # 向下拉菜单中添加 Save 命令
```

```
submenu.add_command(label="Close")                    # 向下拉菜单中添加 Close 命令
menu.add_cascade(label="File", menu=submenu)          # 将下拉菜单添加到菜单中
submenu = tkinter.Menu(menu, tearoff=0)               # 生成下拉菜单
submenu.add_command(label="Copy")                     # 向下拉菜单中添加 Copy 命令
submenu.add_command(label="Paste")                    # 向下拉菜单中添加 Paste 命令
submenu.add_separator()                               # 向下拉菜单中添加分隔符
submenu.add_command(label="Cut")                      # 向下拉菜单中添加 Cut 命令
menu.add_cascade(label="Edit", menu=submenu)          # 将下拉菜单添加到菜单中
submenu = tkinter.Menu(menu, tearoff=0)               # 生成下拉菜单
submenu.add_command(label="About")                    # 向下拉菜单中添加 About 命令
menu.add_cascade(label="Help", menu=submenu)          # 将下拉菜单添加到菜单中
root.config(menu=menu)
root.mainloop()
```

运行脚本后，将创建如图 12-6、图 12-7 和图 12-8 所示的菜单。

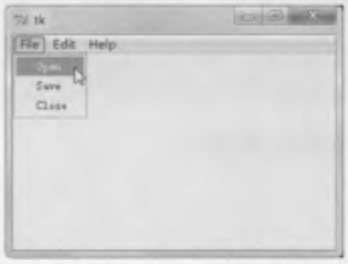


图 12-6 【File】菜单



图 12-7 【Edit】菜单

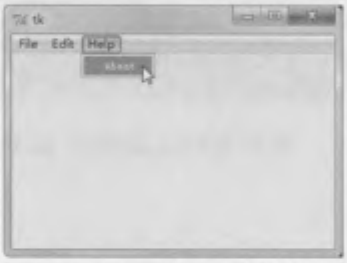


图 12-8 【Help】菜单

创建弹出式菜单与上述创建菜单的过程类似。只需在鼠标单击右键后使用菜单的 post 方法显示菜单即可。下面所示的 tkinterPopupmenu.py 脚本创建了一个简单的右击弹出式菜单。

```
# -*- coding:utf-8 -*-
# file: tkinterPopupmenu.py
#
import tkinter
root = tkinter.Tk()
menu = tkinter.Menu(root, tearoff=0)                  # 创建菜单
menu.add_command(label="Copy")                        # 向弹出式菜单中添加 Copy 命令
menu.add_command(label="Paste")                       # 向弹出式菜单中添加 Paste 命令
menu.add_separator()                                  # 向弹出式菜单中添加分隔符
menu.add_command(label="Cut")                         # 向弹出式菜单中添加 Cut 命令
def popupmenu(event):                                 # 定义右键事件处理函数
    menu.post(event.x_root, event.y_root)             # 显示菜单
root.bind("<Button-3>", popupmenu)                    # 在主窗口中绑定右键事件
root.mainloop()
```

运行 tkinterPopupmenu.py 脚本后将显示一个窗口，在窗口中单击鼠标右键将显示如图 12-9 所示的弹出式菜单。



图 12-9 弹出式菜单

12.2.7 使用单选框和复选框

单选框往往用于一组互斥的选项，即一组单选框中只有一个可以被选中。而复选框则由一个复选框组件来表示两种不同的状态，即被选中表示一种状态，未被选中表示另一种状态。

使用 tkinter.Radiobutton 和 tkinter.Checkbutton 可以分别创建单选框和复选框，通过向其传递参数可以设置单选框和复选框的背景色、大小、状态等。下面是 tkinter.Radiobutton 和 tkinter.Checkbutton 共有的几个控制参数。

- anchor　　指定文本位置。
- background（bg）　　指定背景色。
- borderwidth（bd）　　指定边框的宽度。
- bitmap　　指定组件中的位图。
- font　　指定组件中文本的字体。
- foreground（fg）　　指定组件的前景色。
- height　　指定组件的高度。
- image　　指定组件中的图片。
- justify　　指定组件中多行文本的对齐方式。
- text　　指定组件中的文本，可以使用“\n”表示换行。
- value　　指定组件被选中后关联变量的值。
- variable　　指定组件所关联的变量。
- width　　指定组件的宽度。

对于单选框和复选框，variable 是比较关键的参数。由 variable 指定的变量应使用 tkinter.IntVar 或 tkinter.StringVar 生成。其中，tkinter.IntVar 生成一个整型变量，而 tkinter.StringVar 将生成一个字符串变量。

当使用 tkinter.IntVar 或 tkinter.StringVar 生成变量后，可以使用 set 方法设置变量的初始值。如果该初始值与组件的 value 所指定的值相等，则该组件处于被选中状态。如果其他组件被选中，则其变量值将被更改为该组件 value 所指定的值。

下面所示的 tkinterCheck.py 脚本创建了一组单选框和一个复选框。

```
# -*- coding:utf-8 -*-
# file: tkinterCheck.py
#
import tkinter                          # 导入 tkinter 模块
```

```
root = tkinter.Tk()
r = tkinter.StringVar()                    # 使用 StringVar 生成字符串变量，用于单选框组件
r.set('1')                                 # 初始化变量值
radio = tkinter.Radiobutton(root,          # 生成单选框组件
            variable = r,                  # 设置单选框关联的变量
            value = '1',                   # 设置选中单选框时其所关联的变量的值，即 r 的值
            text = 'Radio1')               # 设置单选框显示的文本
radio.pack()
radio = tkinter.Radiobutton(root,
            variable = r,
            value = '2',                   # 当选中该单选框时 r 的值为 2
            text = 'Radio2' )
radio.pack()
radio = tkinter.Radiobutton(root,
            variable = r,
            value = '3',                   # 当选中该单选框时 r 的值为 3
            text = 'Radio3' )
radio.pack()
radio = tkinter.Radiobutton(root,
            variable = r,
            value = '4',                   # 当选中该单选框时 r 的值为 4
            text = 'Radio4' )
radio.pack()
c = tkinter.IntVar()                       # 使用 IntVar 生成整型变量，用于复选框
c.set(1)
check = tkinter.Checkbutton(root,
            text = 'Checkbutton',          # 设置复选框的文本
            variable = c,                  # 设置复选框关联的变量
            onvalue = 1,                   # 当选中复选框时，c 的值为 1
            offvalue = 2)                  # 当未选中复选框时，c 的值为 2
check.pack()
root.mainloop()
print(r.get())                             # 输出 r 的值
print(c.get())                             # 输出 c 的值
```

运行 tkinterCheck.py 脚本后将创建如图 12-10 所示的单选框和复选框。



图 12-10　单选框和复选框

对于单选框组件和复选框组件还有一个比较特殊的控制参数 indicatoron，当向其传递值 0 时，组件将被绘制成按钮的样式，被选中的组件处于按下状态。下面所示的 tkinterRCButton.py 脚本创建了按钮样式的单选框和复选框。

```
# -*- coding:utf-8 -*-
# file: tkinterRCButton.py
#
```

```
import tkinter                          # 导入 tkinter 模块
root = tkinter.Tk()
r = tkinter.StringVar()                 # 使用 StringVar 生成字符串变量，用于单选框组件
r.set('1')                              # 初始化变量值
radio = tkinter.Radiobutton(root,       # 生成单选框组件
            variable = r,               # 设置单选框关联的变量
            value = '1',                # 设置选中单选框时其所关联的变量的值，即 r 的值
            indicatoron = 0,            # 将单选框绘制成按钮样式
            text = 'Radio1')            # 设置单选框显示的文本
radio.pack()
radio = tkinter.Radiobutton(root,
            variable = r,
            value = '2',                # 当选中该单选框时 r 的值为 2
            indicatoron = 0,
            text = 'Radio2' )
radio.pack()
radio = tkinter.Radiobutton(root,
            variable = r,
            value = '3',                # 当选中该单选框时 r 的值为 3
            indicatoron = 0,
            text = 'Radio3' )
radio.pack()
radio = tkinter.Radiobutton(root,
            variable = r,
            value = '4',                # 当选中该单选框时 r 的值为 4
            indicatoron = 0,
            text = 'Radio4' )
radio.pack()
c = tkinter.IntVar()                    # 使用 IntVar 生成整型变量，用于复选框
c.set(1)
check = tkinter.Checkbutton(root,
            text = 'Checkbutton',       # 设置复选框的文本
            variable = c,               # 设置复选框关联的变量
            indicatoron = 0,            # 将复选框绘制成按钮样式
            onvalue = 1,                # 当选中复选框时，c 的值为 1
            offvalue = 2)               # 当未选中复选框时，c 的值为 2
check.pack()
root.mainloop()
```

运行 tkinterRCButton.py 脚本后将创建如图 12-11 所示的单选框和复选框。



图 12-11 按钮样式的单选框和复选框

12.2.8 绘制图形

使用 tkinter.Canvas 创建 Canvas 绘图组件后，可以使用 Canvas 提供的方法在 Canvas 组件中绘制直线、圆弧、矩形及图片等。Canvas 绘图组件的控制参数主要有以下几个。

- background（bg）　　指定绘图组件的背景色。
- borderwidth（bd）　　指定绘图组件的边框宽度
- bitmap　　指定绘图组件中的位图。
- foreground（fg）　　指定绘图组件的前景色。
- height　　指定绘图组件的高度。
- image　　指定绘图组件中的图片。
- width　　指定绘图组件的宽度。

Canvas 绘图组件的绘图方法主要有以下几种。

- create_arc　　绘制圆弧。
- create_bitmap　　绘制位图，支持 XBM。
- create_image　　绘制图片，支持 GIF。
- create_line　　绘制直线。
- create_oval　　绘制椭圆。
- create_polygon　　绘制多边形。
- create_rectangle　　绘制矩形。
- create_text　　绘制文字。
- create_window　　绘制窗口。
- delete　　删除绘制的图形。

下面所示的 tkinterCanvas.py 脚本是使用 Canvas 绘图组件的绘图方法来绘制不同的图形。

```
# -*- coding:utf-8 -*-
# file: tkinterCanvas.py
#
import tkinter                                    # 导入 tkinter 模块
root = tkinter.Tk()
canvas = tkinter.Canvas(root,
           width = 600,                           # 指定 Canvas 组件的宽度
           height = 480,                          # 指定 Canvas 组件的高度
           bg = 'white')                          # 指定 Canvas 组件的背景色
im = tkinter.PhotoImage(file='python.gif')        # 使用 PhotoImage 打开图片
canvas.create_image(300,50,image = im)            # 使用 create_image 将图片添加到
Canvas 组件中
canvas.create_text(302,77,                        # 使用 create_text 方法绘制文字
       text = 'Use Canvas'                        # 所绘制文字的内容
       ,fill = 'gray')                            # 所绘制文字的颜色为灰色
canvas.create_text(300,75,
       text = 'Use Canvas',
       fill = 'blue')
canvas.create_polygon(290,114,316,114,330,130,    # 使用 create_polygon 方法绘制六边形
           310,146,284,146,270,130)
canvas.create_oval(280,120,320,140,               # 使用 create_oval 方法绘制椭圆
       fill = 'white')                            # 设置椭圆的填充色为白色
```

```
canvas.create_line(250,130,350,130)                    # 使用 create_line 绘制直线
canvas.create_line(300,100,300,160)
canvas.create_rectangle(90,190,510,410,                # 使用 create_rectangle 绘制一个矩形
        width=5)                                       # 设置矩形线宽为 5 个像素
canvas.create_arc(100, 200, 500, 400,                  # 使用 create_arc 绘制圆弧
        start=0, extent=240,                           # 设置圆弧的起止角度
        fill="pink")                                   # 设置圆弧填充颜色
canvas.create_arc(103,203,500,400,
        start=241, extent=118,
        fill="red")
canvas.pack()                                          # 将 Canvas 添加到主窗口
root.mainloop()
```

运行 tkinterCanvas.py 脚本后将创建如图 12-12 所示的窗口。

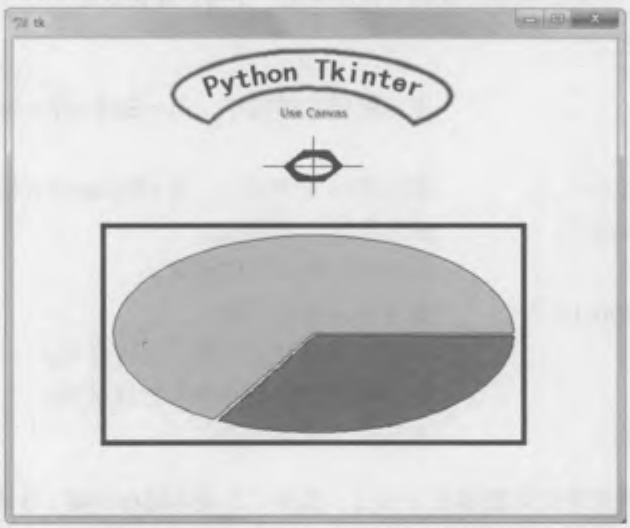


图 12-12 使用 Canvas 绘制的图形

12.3 事件处理

tkinter 中的事件相当于第 11 章中介绍的 MFC 中的消息。对于按钮组件、菜单组件等可以在创建组件时通过 command 参数指定其事件的处理函数。除了组件所触发的事件外，在创建右键弹出菜单时，还需处理右击事件。类似的事件可以归结为鼠标事件、键盘事件和窗口事件。

12.3.1 事件表示

鼠标事件主要是指鼠标按键的按下、释放、鼠标滚轮的滚动，以及鼠标指针移进、移出组件等所触发的事件。键盘事件主要是指键的按下、释放等所触发的事件。窗口事件是指改变窗口大小、组件状态等所触发的事件。

对于鼠标事件、键盘事件和窗口事件，可以采用事件绑定的方法来确定消息的处理方式。事件绑定可以使用组件的 bind 方法进行，或者使用 bind_class 方法进行类绑定，然后分别调用函数或者类来响应事件。bind_all 方法也可以用来绑定事件，bind_all 方法是将所有组件事件绑定到事件

响应函数上。上述三种方法的原型如下。

```
bind(sequence,func,add)
bind_class(className,sequence,func,add)
bind_all(sequence,func,add)
```

各参数含义如下。

- sequence　　所绑定的事件。
- func　　所绑定的事件处理函数。
- add　　可选参数，为空字符或者“+”。
- className　　所绑定的类。

其中，sequence 表示所绑定的事件，必须为以“<>”包围的字符串。对于鼠标事件则可以使用以下几种表示方式。

- <Button-1>　　表示鼠标左键按下，而<Button-2>表示中键，<Button-3>表示右键。
- <ButtonPress-1>　　表示鼠标左键按下，与<Button-1>相同。
- <ButtonRelease-1>　　表示鼠标左键释放。
- <B1-Motion>　　表示按住鼠标左键移动。
- <Double-Button-1>　　表示双击鼠标左键。
- <Enter>　　表示鼠标指针进入某一组件区域。
- <Leave>　　表示鼠标指针离开某一组件区域。
- <MouseWheel>　　表示鼠标滚轮动作。

上述鼠标事件中的数字均可替换成 2 或 3，其中，2 表示鼠标中键，3 表示鼠标右键。例如，<B3-Motion>表示按住鼠标右键移动，<Double-Button-2>表示双击鼠标中键等。

对于键盘事件则可以使用以下几种表示方式。

- <KeyPress-A>　　表示按下 A 键，可用其他字母键代替。
- <Alt-KeyPress-A>　　表示同时按下 Alt 键和 A 键。
- <Control-KeyPress-A>　　表示同时按下 Control 键和 A 键。
- <Shift-KeyPress-A>　　表示同时按下 Shift 键和 A 键。
- <Double-KeyPress-A>　　表示快速地按两下 A 键。
- <Lock-KeyPress-A>　　表示 Caps Lock 打开后按下 A 键。

上述键盘事件还可以使用 Alt、Control 和 Shift 键组合。例如，<Alt-Control-Shift-KeyPress-B>表示同时按下 Alt、Control、Shift 和 B 键。其中，KeyPress 可以用 KeyRelease 替换，表示当按键释放时触发事件。需要注意的是，如果输入的是字母则要区分大小写，如果使用<KeyPress-A>，则只有按下 Shift 键或者 Caps Lock 键打开时才触发事件。

对于窗口事件则可以使用以下几种表示方法。

- Activate　　当组件由不可用转为可用时触发。

- Configure　当组件大小改变时触发。
- Deactivate　当组件由可用转为不可用时触发。
- Destroy　当组件被销毁时触发。
- Expose　当组件从被遮挡状态中暴露出来时触发。
- FocusIn　当组件获得焦点时触发。
- FocusOut　当组件失去焦点时触发。
- Map　当组件由隐藏状态变为显示状态时触发。
- Property　当窗体的属性被删除或改变时触发。
- Unmap　当组件由显示状态变为隐藏状态时触发。
- Visibility　当组件变为可视状态时触发。

12.3.2 响应事件

窗体中的事件被绑定到函数后，当该事件被触发后将调用所绑定的函数进行处理。事件触发后系统将向该函数传递一个 event 对象的参数。因此被绑定的响应事件的函数应该定义成下面所示的形式。

```
def function(event):
    <语句>
```

其中，event 对象具有以下属性。

- char　按键字符，仅对键盘事件有效。
- keycode　按键名，仅对键盘事件有效。
- keysym　按键编码，仅对键盘事件有效。
- num　鼠标按键，仅对鼠标事件有效。
- type　所触发的事件类型。
- widget　引起事件的组件。
- width, height　组件改变后的大小，仅对 Configure 有效。
- x, y　鼠标当前位置，相对于窗口。
- x_root, y_root　鼠标当前位置，相对于整个屏幕。

下面所示的 tkinterDraw.py 脚本是使用事件处理来创建一个简单的绘图程序。在脚本中，主要使用了 events 对象的 x 和 y 属性来绘制图形。

```
# -*- coding:utf-8 -*-
# file: tkinterDraw.py
#
import tkinter                                          # 导入 tkinter 模块
class MyButton:                                         # 定义按钮类
    def __init__(self,root,canvas,label,type):         # 类初始化
        self.root = root                                # 保存引用值
        self.canvas = canvas
        self.label = label
        if type == 0:                                   # 根据类型创建按钮
```

```
            button = tkinter.Button(root,text = 'DrawLine',
                    command = self.DrawLine)
        elif type == 1:
            button = tkinter.Button(root,text = 'DrawArc',
                    command = self.DrawArc)
        elif type == 2:
            button = tkinter.Button(root,text = 'DrawRec',
                    command = self.DrawRec)
        else :
            button = tkinter.Button(root,text = 'DrawOval',
                    command = self.DrawOval)
        button.pack(side = 'left')
    def DrawLine(self):                                    # DrawLine 按钮事件处理函数
        self.label.text.set('Draw Line')
        self.canvas.SetStatus(0)
    def DrawArc(self):                                     # DrawArc 按钮事件处理函数
        self.label.text.set('Draw Arc')
        self.canvas.SetStatus(1)
    def DrawRec(self):                                     # DrawRec 按钮事件处理函数
        self.label.text.set('Draw Rectangle')
        self.canvas.SetStatus(2)
    def DrawOval(self):                                    # DrawOval 按钮事件处理函数
        self.label.text.set('Draw Oval')
        self.canvas.SetStatus(3)
class MyCanvas:                                            # 定义 Canvas 类
    def __init__(self,root):
        self.status = 0                                    # 保存引用值
        self.draw = 0
        self.root = root
        self.canvas = tkinter.Canvas(root,bg = 'white',    # 生成 Canvas 组件
                width = 600,
                height = 480)
        self.canvas.pack()
        self.canvas.bind('<ButtonRelease-1>',self.Draw)    # 绑定事件到左键
        self.canvas.bind('<Button-2>',self.Exit)           # 绑定事件到中键
        self.canvas.bind('<Button-3>',self.Del)            # 绑定事件到右键
        self.canvas.bind_all('<Delete>',self.Del)          # 绑定事件到 Delete 键
        self.canvas.bind_all('<KeyPress-d>',self.Del)      # 绑定事件到 d 键
        self.canvas.bind_all('<KeyPress-e>',self.Exit)     # 绑定事件到 e 键
    def Draw(self,event):                                  # 绘图事件处理函数
        if self.draw == 0:                                 # 判断是否绘图
            self.x = event.x
            self.y = event.y
            self.draw = 1
        else:                                              # 根据 self.status 绘制不同图形
            if self.status == 0:
                self.canvas.create_line(self.x,self.y,
                        event.x,event.y)
                self.draw = 0
            elif self.status == 1:
                self.canvas.create_arc(self.x,self.y,
                        event.x,event.y)
                self.draw = 0
            elif self.status == 2:
                self.canvas.create_rectangle(self.x,self.y,
```

```
                    event.x,event.y)
                self.draw = 0
            else:
                self.canvas.create_oval(self.x,self.y,
                        event.x,event.y)
                self.draw = 0
    def Del(self,event):                                    # 当按下右键或 d 键时删除图形
        items = self.canvas.find_all()
        for item in items:
            self.canvas.delete(item)
    def Exit(self,event):                                   # 当按下中键或 e 键时退出
        self.root.quit()
    def SetStatus(self,status):                             # 设置绘制的图形
        self.status = status
class MyLabel:                                              # 定义标签类
    def __init__(self,root):                                # 类初始化
        self.root = root                                    # 保存引用
        self.canvas = canvas
        self.text = tkinter.StringVar()                     # 生成标签引用变量
        self.text.set('Draw Line')
        self.label = tkinter.Label(root,textvariable = self.text,  # 生成标签
                fg = 'red',width = 50)
        self.label.pack(side = 'left')
root = tkinter.Tk()                                         # 生成主窗口
canvas = MyCanvas(root)                                     # 生成绘图组件
label = MyLabel(root)                                       # 生成标签
MyButton(root,canvas,label,0)                               # 生成按钮
MyButton(root,canvas,label,1)
MyButton(root,canvas,label,2)
MyButton(root,canvas,label,3)
root.mainloop()                                             # 进入消息循环
```

运行 tkinterDraw.py 脚本后将创建如图 12-13 所示的主窗口。在窗口中单击鼠标左键，然后移动到另一位置再单击左键将绘制图形。可以单击按钮选择要绘制的图形。单击右键或者按键盘上的 D 键将删除多绘制的图形。单击鼠标中键或者按键盘上的 E 键将关闭窗口。

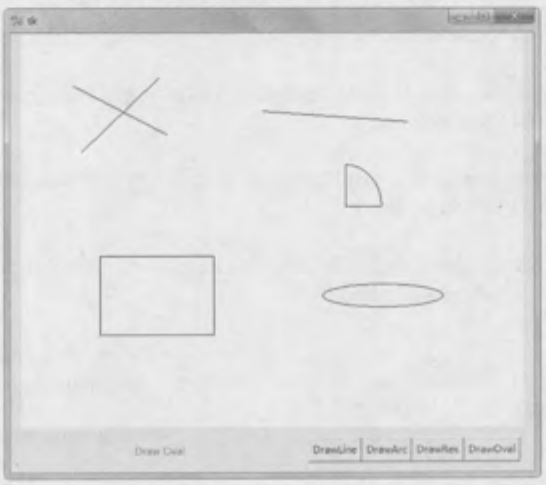


图 12-13 主窗口

12.4 创建对话框

tkinter 中提供了标准的对话框，在脚本中可以直接使用这些标准对话框与用户交互。如果 tkinter 提供的对话框不能满足要求，则还可以使用 Toplevel 来创建对话框。

12.4.1 使用标准对话框

标准对话框包含简单的消息框和用户输入对话框。其中，信息框以窗口的形式向用户输出信息，也可以获取用户所单击的按钮。输入对话框要求用户输入字符串、整型或者浮点型的值。

1. 消息框

tkinter.messagebox 模块提供了几种简单的消息框（在 Python 2.x 中，由 tkMessageBox 模块提供）。使用 tkinter.messagebox 模块中的 askokcancel、askquestion、askyesno、showerror、showinfo 和 showwarning 可以创建简单的消息框。使用这些函数时只需向其传递 title 和 message 参数即可。

下面所示的 tkinterMessageBox.py 脚本是使用 tkMessageBox 来创建简单的消息框。

```
# -*- coding:utf-8 -*-
# file: tkinterMessageBox.py
#
import tkinter                                          # 导入 tkinter 模块
import tkinter.messagebox                               # 导入 tkMessageBox 模块
def cmd():                                              # 按钮消息处理函数
    global n                                            # 使用全局变量 n
    global buttontext                                   # 使用全局变量 buttontext
    n = n + 1
    if n == 1:                                          # 判断 n 的值，显示不同的消息框
        tkinter.messagebox.askokcancel('Python tkinter','askokcancel')     # 使
用 askokcancel 函数
        buttontext.set('skquestion')                    # 更改按钮上的文字
    elif n == 2:
        tkinter.messagebox.askquestion('Python tkinter','skquestion') # 使用
askquestion 函数
        buttontext.set('askyesno')
    elif n == 3:
        tkinter.messagebox.askyesno('Python tkinter','askyesno') # 使用 askyesno 函数
        buttontext.set('showerror')
    elif n == 4:
        tkinter.messagebox.showerror('Python tkinter','showerror')  # 使用 showerror 函数
        buttontext.set('showinfo')
    elif n == 5:
        tkinter.messagebox.showinfo('Python tkinter','showinfo')  # 使用 showinfo 函数
        buttontext.set('showwarning')
    else :
        n = 0                                           # 将 n 赋值为 0 重新开始循环
        tkinter.messagebox.showwarning('Python tkinter','showwarning')    # 使用 showwarning 函数
        buttontext.set('askokcancel')
n = 0                                                   # 为 n 赋初始值
root = tkinter.Tk()
buttontext = tkinter.StringVar()                        # 生成关联按钮文字的变量
```

```
buttontext.set('askokcancel')                    # 设置 buttontext 值
button = tkinter.Button(root,                    # 生成按钮
        textvariable = buttontext,               # 设置关联变量
        command = cmd)                           # 设置事件处理函数
button.pack()
root.mainloop()                                  # 进入消息循环
```

运行 tkinterMessageBox.py 脚本后，单击主窗口的按钮，将依次创建如图 12-14、图 12-15、图 12-16、图 12-17、图 12-18 和图 12-19 所示的消息框。

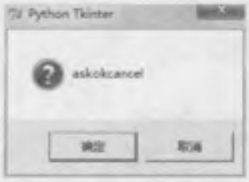


图 12-14 askokcancel 消息框



图 12-15 askquestion 消息框



图 12-16 askyesno 消息框



图 12-17 showerror 消息框



图 12-18 showinfo 消息框



图 12-19 showwarning 消息框

除了上述 6 个标准消息框外，还可以使用 tkinter.messagebox._show 函数创建其他类型的消息框。tkinter.messagebox._show 函数的控制参数如下。

- default　　指定消息框的按钮。
- icon　　指定消息框的图标。
- message　　指定消息框所显示的消息。
- parent　　指定消息框的父组件。
- title　　指定消息框的标题。
- type　　指定消息框的类型。

2. 使用标准对话框

使用 tkinter.simpledialog 模块、tkinter.filedialog 模块、tkinter.colorchooser 模块（在 Python 2.x 中是 tkSimpleDialog 模块、tkFileDialog 模块和 tkColorChooser 模块）可以创建标准的对话框。其中，tkinter.simpledialog 模块可以创建标准的输入对话框。tkinter.filedialog 模块可以创建文件打开和保存文件对话框。tkinter.colorchooser 模块可以创建颜色选择对话框。

tkinter.simpledialog 模块可以创建三种类型的对话框：输入字符串、输入整数、输入浮点型。其对应函数分别为 askstring、askinteger 和 askfloat。其具有以下几个相同的可选参数。

- title　　指定对话框标题。
- prompt　　指定对话框中显示的文字。
- initialvalue　　指定输入框的初始值。

使用 tkinter.simpledialog 模块中的函数创建对话框后，将返回对话框中文本框的值。下面所示

的 tkinterSimpleDialog.py 脚本分别创建了 3 种简单的输入对话框。

```
# -*- coding:utf-8 -*-
# file: tkinterSimpleDialog.py
#
import tkinter                                          # 导入 tkinter 模块
import tkinter.simpledialog                             # 导入 tkSimpleDialog 模块
def InStr():                                            # 按钮事件处理函数
    r = tkinter.simpledialog.askstring('Python tkinter',   # 创建字符串输入对话框
            'Input String',                             # 指定提示字符
            initialvalue='tkinter')                     # 指定初始化文本
    print(r)                                            # 输出返回值
def InInt():                                            # 按钮事件处理函数
    r = tkinter.simpledialog.askinteger('Python tkinter','Input Integer') # 创
建整数输入对话框
    print(r)
def InFlo():                                            # 按钮事件处理函数
    r = tkinter.simpledialog.askfloat('Python tkinter','Input Float') # 创建浮点
数输入对话框
    print(r)
root = tkinter.Tk()
button1 = tkinter.Button(root,text = 'Input String',   # 创建按钮
       command = InStr)                                 # 指定按钮事件处理函数
button1.pack(side='left')
button2 = tkinter.Button(root,text = 'Input Integer',
       command = InInt)                                 # 指定按钮事件处理函数
button2.pack(side='left')
button2 = tkinter.Button(root,text = 'Input Float',
       command = InFlo)                                 # 指定按钮事件处理函数
button2.pack(side='left')
root.mainloop()                                         # 进入消息循环
```

运行 tkinterSimpleDialog.py 脚本后将创建如图 12-20 所示的主窗口，分别单击 3 个按钮将创建如图 12-21、图 12-22 和图 12-23 所示的输入对话框。



图 12-20　创建的主窗口

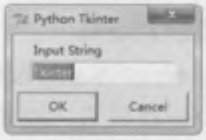


图 12-21　【输入字符串】对话框

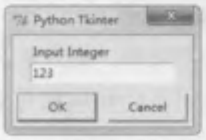


图 12-22　【输入整数】对话框

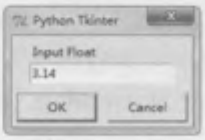


图 12-23　【输入浮点数】对话框

tkinter.filedialog 模块中的 askopenfilename 函数可以创建标准的【打开文件】对话框。asksaveasfilename 可以创建标准的【保存文件】对话框。其具有以下几个相同的可选参数。

- filetypes　　指定文件类型。

- initialdir　　指定默认目录。
- initialfile　　指定默认文件。
- title　　指定对话框标题。

使用 tkFileDialog 模块中的函数创建对话框后，将返回文件的完整路径。下面所示的 tkinterFileDialog.py 脚本创建了文件打开和保存对话框。

```
# -*- coding:utf-8 -*-
# file: tkinterFileDialog.py
#
import tkinter                                          # 导入 tkinter 模块
import tkinter.filedialog                               # 导入 tkFileDialog 模块
def FileOpen():                                         # 按钮事件处理函数
    r = tkinter.filedialog.askopenfilename(title = 'Python tkinter',  # 创建打开
文件对话框
            filetypes=[('Python', '*.py *.pyw'),('All files', '*')]    )
    # 指定文件类型为 Python 脚本
    print(r)                                            # 输出返回值
def FileSave():                                         # 按钮事件处理函数
    r = tkinter.filedialog.asksaveasfilename(title = 'Python tkinter',# 创建保存
文件对话框
            initialdir=r'E:\Python\code',               # 指定初始化目录
            initialfile = 'test.py')                    # 指定初始化文件
    print(r)
root = tkinter.Tk()
button1 = tkinter.Button(root,text = 'File Open',       # 创建按钮
       command = FileOpen)                              # 指定按钮事件处理函数
button1.pack(side='left')
button2 = tkinter.Button(root,text = 'File Save',
       command = FileSave)                              # 指定按钮事件处理函数
button2.pack(side='left')
root.mainloop()                                         # 进入消息循环
```

运行 tkinterFileDialog.py 脚本后，单击【File Open】按钮，将创建如图 12-24 所示的【打开文件】对话框；单击【File Save】按钮，将创建如图 12-25 所示的【保存文件】对话框。

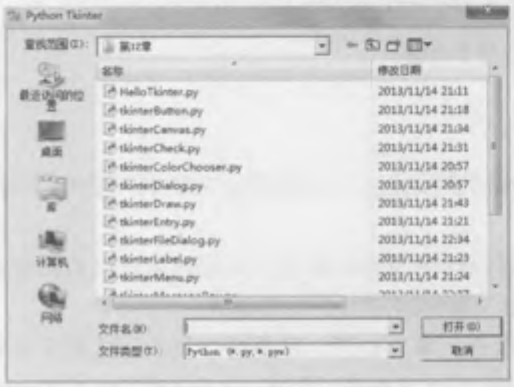


图 12-24　【打开文件】对话框

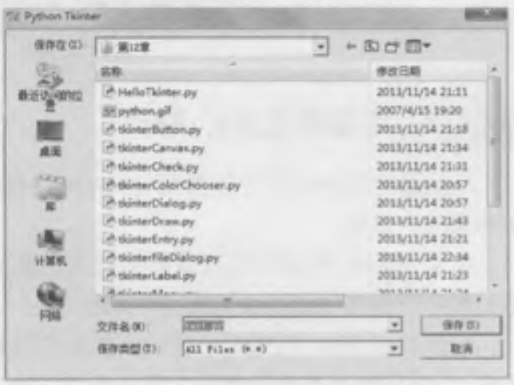


图 12-25　【保存文件】对话框

tkinter.colorchooser 模块中的 askcolor 函数可以创建标准的【颜色选择】对话框，其具有以下几个可选参数。

- initialcolor　　指定初始化颜色。
- title　　指定对话框标题。

使用 tkinter.colorchooser 模块中的函数创建对话框后，将返回颜色的 RGB 值以及可以在 Python tkinter 中使用的颜色字符值。下面所示的 tkinterColorChooser.py 脚本创建了【颜色选择】对话框。

```
# -*- coding:utf-8 -*-
# file: tkinterColorChooser.py
#
import tkinter                                          # 导入 tkinter 模块
import tkinter.colorchooser                             # 导入 tkColorChooser 模块
def ChooseColor():                                      # 按钮事件处理函数
    r = tkinter.colorchooser.askcolor(title = 'Python tkinter') # 创建颜色选择对话框
    print(r)                                            # 输出返回值
root = tkinter.Tk()
button = tkinter.Button(root,text = 'Choose Color',     # 创建按钮
       command = ChooseColor)                           # 指定按钮事件处理函数
button.pack()
root.mainloop()                                         # 进入消息循环
```

运行 tkinterColorChooser.py 脚本后，单击【Choose Color】命令，将创建如图 12-26 所示的【颜色选择】对话框。

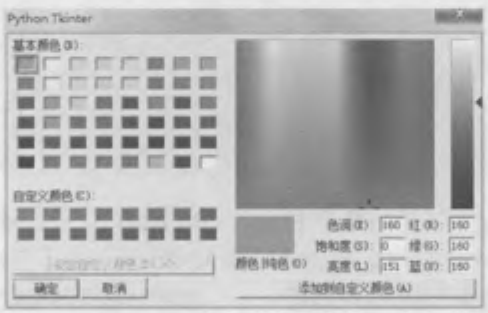


图 12-26　【颜色选择】对话框

12.4.2 创建自定义对话框

从前面介绍的内容可以看出，tkinter 中提供了简单的对话框，可以方便地在脚本中使用，以打开标准的对话框。

如果 tkinter 所提供的对话框不能满足要求，则可以使用 Toplevel 组件来创建自定义对话框。在脚本中可以向 Toplevel 组件添加其他组件，并且定义事件响应函数或者类等。

在使用 tkinter 创建对话框时，如果对话框中需要进行事件处理，则最好以类的形式来定义对话框，否则只能大量使用全局变量来处理参数，这会导致脚本维护调试困难。对于代码较多的 tkinter GUI Python 脚本，整个脚本也应该使用类的方式来组织。

下面所示的 tkinterDialog.py 脚本是使用 Toplevel 组件来创建一个简单的对话框。

```
# -*- coding:utf-8 -*-
# file: tkinterDialog.py
#
import tkinter                                          # 导入 tkinter 模块
import tkinter.messagebox                               # 导入 tkMesageBox 模块
class MyDialog:                                         # 定义对话框类
    def __init__(self, root):                           # 对话框初始化
        self.top = tkinter.Toplevel(root)               # 生成 Toplevel 组件
        label = tkinter.Label(self.top, text='Please Input')   # 生成标签组件
        label.pack()
        self.entry = tkinter.Entry(self.top)            # 生成文本框组件
        self.entry.pack()
        self.entry.focus()                              # 让文本框获得焦点
        button = tkinter.Button(self.top, text='Ok',    # 生成按钮
                command=self.Ok)                        # 设置按钮事件处理函数
        button.pack()
    def Ok(self):                                       # 定义按钮事件处理函数
        self.input = self.entry.get()                   # 获取文本框中的内容,保存为 input
        self.top.destroy()                              # 销毁对话框
    def get(self):                                      # 返回在文本框输入的内容
        return self.input
class MyButton():                                       # 定义按钮类
    def __init__(self, root, type):                     # 按钮初始化
        self.root = root                                # 保存父窗口引用
        if type == 0:                                   # 根据类型创建不同按钮
            self.button = tkinter.Button(root,
                    text='Create',
                    command = self.Create)              # 设置 Create 按钮的事件处理函数
        else:
            self.button = tkinter.Button(root,
                    text='Quit',
                    command = self.Quit)                # 设置 Quit 按钮的事件处理函数
        self.button.pack()
    def Create(self):                                   # Create 按钮的事件处理函数
        d = MyDialog(self.root)                         # 生成对话框
        self.button.wait_window(d.top)                  # 等待对话框结束
        tkMessageBox.showinfo('Python','You input:\n' + d.get())
                                                        # 获取对话框中输入值，并输出
    def Quit(self):                                     # Quit 按钮的事件处理函数
        self.root.quit()                                # 退出主窗口
root = tkinter.Tk()                                     # 生成主窗口
MyButton(root,0)                                        # 生成 Create 按钮
MyButton(root,1)                                        # 生成 Quit 按钮
root.mainloop()                                         # 进入消息循环
```

运行 tkinterDialog.py 脚本后，将创建如图 12-27 所示的窗口，单击 Create 按钮后，将创建如图 12-28 所示的对话框，在其中的文本框中输入一些文字，单击【OK】按钮后，将弹出如图 12-29

所示的信息框。在图 12-27 所示窗口中单击【Quit】按钮，则会退出窗口。



图 12-27　创建的主窗口

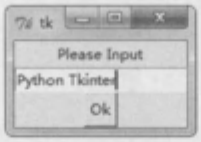


图 12-28　创建的对话框



图 12-29　消息框

12.5 本章小结

本章首先介绍了 tkinter 模块的一些基本概念，接着介绍了 tkinter 模块中常用组件的创建方法和这些组件的常用属性，然后介绍了 tkinter 模块事件处理的方法，最后介绍了使用 tkinter 标准对话框及创建自定义对话框的方法。

下一章将学习另一种 GUI 设计模块：wxPython。

◆　　◆　　◆

第 13 章　使用 wxPython 编写 GUI

本章包括

- wxPython 概述
- 使用 wxPython 消息框和标准对话框
- 创建菜单
- 用 wxPython 创建文本编辑器
- 使用 wxPython 组件
- 创建自定义对话框
- 绑定菜单事件

wxPython 是跨平台 GUI 工具库 wxWidgets 的封装。wxWidgets 库是由 C++编写的，类似于 Winodws 的 MFC。wxWidgets 提供了对多种操作系统的支持，因此 wxWidgets 具有良好的可移植性，wxPython 具备跨平台的能力。在 Python 中，使用 wxPython 可以编写具有跨平台能力的 GUI 脚本。

13.1　wxPython 概述

与 Tkinter 不同，wxPython 需要用户自己安装，不过安装过程并不复杂。wxPython 的使用也不复杂，有过 MFC 编程经验的读者可以很快熟悉 wxPython。

13.1.1　安装 wxPython

由于 wxPython 不是作为 Python 的一部分随官方发布的，因此需要从其官方网站 http://www.wxpython.org/下载安装。根据所使用的 Python 版本，下载相应的安装程序，同时需要注意，wxPython 安装程序分为 Unicode 版和 Ansi 版。其中，Unicode 版提供了对 Unicode 的支持，如果在程序中需要使用中文，则应该下载 Unicode 版的 wxPython。

需要注意的是，目前 wxPython 官方网站上只有针对 Python 2.x 的安装版本，图 13-1 所示的是 wxPython 官网下载页显示的内容。

如果正在使用的是 Python 2.x，则可以直接下载对应版本的 wxPython 安装程序，然后进行安装使用。

对于 Python 3 用户来说，要想使用 wxPython 则需要采取另一种方式（相信适用于 Python 3 的 wxPython 安装程序很快会出来），并且按以下步骤进行操作。

step 1　在网页浏览器中输入网址 http://wxpython.org/Phoenix/snapshot-builds/，可看到一个很长的列表，其中列出了适用于不同版本的 wxPython 压缩包。

step 2　根据需要下载一个压缩包（例如，下载 wxPython_Phoenix-3.0.0.0-r75126-win32-py3.2.tar.gz，表示适用于 Windows 平台 Python 3.2 版本的 wxPython）。

step 3　将下载的压缩包进行解压缩操作，在解压的目录中可以看到一个名为 wx 的目录。将这个名为 wx 的目录复制到 C:\Python32\Lib\site-packages\目录中。

通过以上 3 步，即可在 Python 中使用 wxPython 了。首先，可以在 Python 交互式环境下输入以下语句进行测试。

```
import wx
```

如果没有出现运行错误，则表示 wxPython 安装成功。可以使用 wxPython 进行 GUI 编程了。

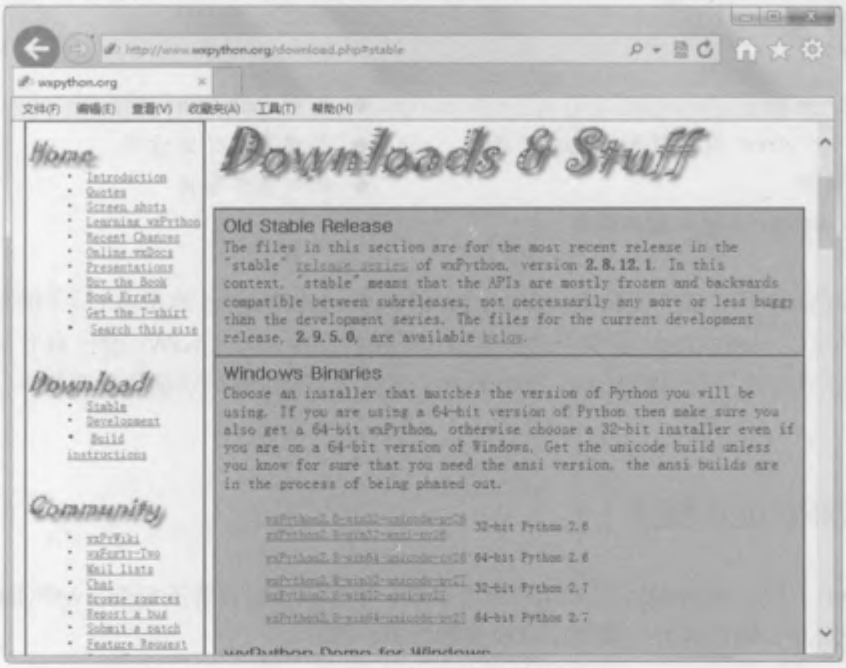


图 13-1　wxPython 官网下载页面

13.1.2　创建窗口

wxPython 是对 wxWidgets C++类库的封装，因此，其使用方法类似于 Windows 的 MFC，即在脚本中应该继承 wxPython 中相应的类。也就是说，在使用 wxPython 的 Python 脚本时，应该首先继承 wxPython 中的 wxApp 类。下面所示的 HellowxPython.py 脚本是使用 wxPython 创建一个简单的窗口。

```
# -*- coding:utf-8 -*-
# file: HellowxPython.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None,title = 'Hello,wxPython!')
        frame.Show()
        return True
app = MyApp()
app.MainLoop()
```

运行 HellowxPython.py 脚本后，将创建如图 13-2 所示的窗口。



图 13-2 wxPython 窗口

在上述的 HellowxPython.py 脚本中，首先导入了 wx 模块（即 wxPython），然后继承 wx 模块中的 App 类，创建了一个 MyApp 类，并在 MyApp 类中重载了 OnInit 方法。OnInit 方法是作为窗口初始化的一部分。在 OnInit 方法中创建一个窗口框架，并显示该窗口。一般来说，OnInit 方法最后应返回 True。在脚本的最后两行，将 MyApp 类实例化成 app 对象，然后调用其 MainLoop 方法进入消息循环。

上述脚本代码量很少，但是即使是更复杂的脚本，也遵循这样的过程。复杂的脚本无非是在 OnInit 方法中做更多的初始化工作，定义事件响应函数等，最终还是要进入消息循环。

下面所示的 SimpleHello.py 脚本，则以更加简洁的形式创建了一个窗口。

```
# -*- coding:utf-8 -*-
# file: SimpleHello.py
#
import wx
app = wx.PySimpleApp()
frame = wx.Frame(parent = None,title = 'Simple Hello')
frame.Show(True)
app.MainLoop()
```

运行 SimpleHello.py 脚本后，将创建如图 13-3 所示的 Simple Hello 窗口。



图 13-3 Simple Hello 窗口

在 SimpleHello.py 脚本中，没有继承 wx 的 App 类，而仅使用了 PySimpleApp 生成的一个实例对象，然后再创建一个框架窗口，最后进入消息循环。该方法适用于简单窗口的创建，省去了定

义类的过程。

13.2 组件

在 wxPython 中，只要使用相应的组件类即可创建组件，非常简便快捷，wxPython 提供了丰富的组件用于完成 GUI 程序的创建。在 wxPython 中，所创建的组件应该被包含在框架窗口或者面板中，13.1 节介绍了创建窗口的代码，本节将首先介绍创建面板的方法，然后介绍如何在窗口或面板中添加各类组件。

13.2.1 面板

在 wxPython 中，框架窗口可以容纳其他组件。但一般情况下，框架窗口仅包含面板组件，面板组件作为“容器”，在其中添加其他的组件。

使用 wx.Panel 类可以创建一个面板组件，wx.Panel 具有以下初始化参数，这些参数中除 parent 参数外，其他都是可选参数。

- parent　　面板的父组件，一般为所创建的框架窗口。
- id　　面板的 ID，如果不指定可以将其设为-1。
- pos　　面板在父组件中的位置。
- size　　面板的大小。
- style　　面板的样式。
- name　　面板的名字。

下面所示的 wxPythonPanel.py 脚本是向框架窗口中添加面板。

```
# -*- coding:utf-8 -*-
# file: wxPythonPanel.py
#
import wx                                                    # 导入 wxPython 模块
class MyApp(wx.App):                                         # 通过继承 wx.App 类创建类
    def OnInit(self):                                        # 重载 OnInit 方法
        frame = wx.Frame(parent = None,                      # 创建框架窗口
                id=-1,                                       # 指定框架 ID
                title='Panel',                               # 指定窗口标题
                pos=(100,100),                               # 指定窗口位置
                size=(600,480),                              # 指定窗口大小
                style=wx.DEFAULT_FRAME_STYLE,                # 指定窗口样式
                name="frame")                                # 指定窗口名
        panel = wx.Panel(frame, -1)                          # 向框架窗口添加面板
        frame.Show()                                         # 显示框架窗口
        return True                                          # 返回 True
app = MyApp()                                                # 类实例化
app.MainLoop()                                               # 进入消息循环
```

wxPythonPanel.py 脚本中使用了完整的 wx.Frame 初始化参数，除了 parent 参数外，其他都是

可以省略的。运行 wxPythonPanel.py 脚本后，将创建如图 13-4 所示的窗口。



图 13-4 向框架窗口中添加面板

13.2.2 按钮

在 wxPython 中，可以使用 wx.Button 类创建按钮，当按钮创建后就可以绑定按钮事件，并定义响应按钮事件的函数。

1. 创建按钮

使用 wx.Button 创建按钮时，可以向其传递如下所示的初始化参数。

- parent 包含该按钮的父组件。
- id 按钮的 ID。
- label 按钮所显示的文字。
- pos 按钮的位置。
- size 按钮的大小。
- style 按钮的样式。
- validator 按钮的验证类。
- name 按钮名。

下面所示的 wxPythonButton.py 脚本是使用 wx.Button 创建按钮。

```
# -*- coding:utf-8 -*-
# file: wxPythonButton.py
#
import wx                                                    # 导入 wxPython
class MyApp(wx.App):                                         # 通过继承创建类
    def OnInit(self):                                        # 重载 OnInit 方法
        frame = wx.Frame(parent = None,title = 'Button')     # 生成框架窗口
        panel = wx.Panel(frame, -1)                          # 生成面板
        button = wx.Button(panel,                            # 向面板添加按钮
                -1,                                          # 指定按钮 ID
                'Button',                                    # 指定按钮上的文本
                pos=(50,50))                                 # 指定按钮在面板上的位置
```

```
        frame.Show()                                    # 显示窗口
        return True
app = MyApp()                                           # 类实例化
app.MainLoop()                                          # 进入消息循环
```

运行 wxPythonButton.py 脚本后，将创建如图 13-5 所示的窗口。



图 13-5　创建按钮

2. 按钮事件处理

如果要对按钮绑定事件，则可以使用 wx.App 类的 Bind 方法，其原型如下。

```
Bind(event, handler, source=None, id=-1, id2=-1)
```

其参数含义如下。

- event　　所绑定的事件类型。
- handler　　事件的响应函数。
- source　　事件源。
- id　　用于使用组件 ID 代替事件源。
- id2　　用于指定多个组件。

其中事件的响应函数应定义成如下所示的形式。

```
def OnEvent(self, event):
    <处理语句>
```

事件响应函数将接受一个 event 对象的参数，不同的事件，将接受不同的 event 对象。下面所示的 wxPythonButtonEvent.py 脚本是使用 wx.App 类的 Bind 方法绑定按钮事件。

```
# -*- coding:utf-8 -*-
# file: wxPythonButtonEvent.py
#
import wx                                               # 导入 wxPython
class MyApp(wx.App):                                    # 通过继承创建类
    def OnInit(self):                                   # 重载 OnInit 方法
```

```
        frame = wx.Frame(parent = None,title = 'wxPython',size = (300,170))
                                                          # 生成框架窗口
        panel = wx.Panel(frame, -1)                       # 生成面板
        self.button1 = wx.Button(panel, -1, 'Button1', pos=(50, 50)) # 添加Button1
        self.Bind(wx.EVT_BUTTON,                          # 绑定按钮事件
            self.OnButton1,                               # 指定事件响应函数
            self.button1)                                 # 指定按钮
        self.button2 = wx.Button(panel, -1, 'Button2',pos = (150,50))
        self.Bind(wx.EVT_BUTTON,                          # 绑定按钮事件
            self.OnButton2,                               # 指定事件响应函数
            self.button2)                                 # 指定按钮
        self.button1.SetDefault()                         # 将Button1设为默认按钮
        frame.Show()                                      # 显示窗口
        return True
    def OnButton1(self, event):                           # 按钮事件响应函数
        self.button2.SetLabel('Button1')                  # 更改Button2的文字
        self.button2.SetDefault()                         # 将Button2设为默认按钮
        self.button1.SetLabel('Button2')                  # 更改Button1的文字
    def OnButton2(self, event):                           # 按钮事件响应函数
        self.button1.SetLabel('Button1')                  # 更改Button1的文字
        self.button1.SetDefault()                         # 将Button1设为默认按钮
        self.button2.SetLabel('Button2')                  # 更改Button2的文字
app = MyApp()
app.MainLoop()
```

运行 wxPythonButtonEvent.py 脚本，将显示如图 13-6 左图所示窗口，在左图中单击【Button1】按钮后，两个按钮显示的内容将互换，得到图 13-6 中右图所示结果。

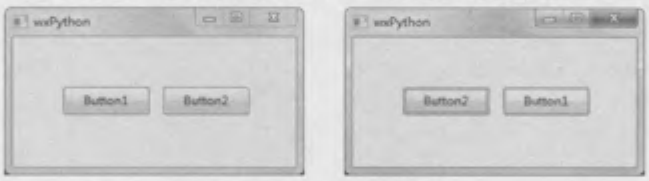


图 13-6 绑定按钮事件

13.2.3 标签

在 wxPython 中使用 wx.StaticText 类可以创建标签。其初始化参数如下。

- parent 包含该标签的父组件。
- id 标签的 ID。
- label 标签所显示的文本。
- pos 标签的位置。
- size 标签的大小。
- style 标签的样式。
- name 标签名。

下面所示的 wxPythonStatic.py 脚本是使用 wx.StaticText 类创建标签。

```
# -*- coding:utf-8 -*-
# file: wxPythonStatic.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None,title = 'wxPython',size = (300,200))
    # 生成框架窗口
        panel = wx.Panel(frame, -1)                          # 生成面板
        label1 = wx.StaticText(panel,                        # 生成标签
                -1,                                          # 指定标签 ID
                'Python',                                    # 指定标签中文本
                size = (160,20),                             # 指定标签大小
                pos = (60,10),                               # 指定标签位置
                style = wx.ALIGN_RIGHT)                      # 指定标签样式，右对齐
        label2 = wx.StaticText(panel,                        # 生成标签
                -1,                                          # 指定标签 ID
                'Python',                                    # 指定标签中文本
                size = (160,20),                             # 指定标签大小
                pos = (60,50),                               # 指定标签位置
                style = wx.ALIGN_CENTER)                     # 指定标签样式，居中对齐
        label2.SetForegroundColour('red')                    # 指定标签前景色
        label2.SetBackgroundColour('black')                  # 指定标签背景色
        label3 = wx.StaticText(panel,                        # 生成标签
                -1,                                          # 指定标签 ID
                'Python\nwxPython',                          # 在文本中使用换行符
                size = (160,40),                             # 指定标签大小
                pos = (60,90))                               # 指定标签位置
        frame.Show()
        return True
app = MyApp()
app.MainLoop()
```

运行 wxPythonStatic.py 脚本后，将创建如图 13-7 所示的窗口。



图 13-7 创建标签

13.2.4 文本框

在 wxPython 中使用 wx.TextCtrl 类可以创建文本框，通过改变其样式可以创建单行文本和多行文本。另外，在 wxPython 中还有一类可以设置颜色和字体的文本框。

1. 单行文本框

使用 wx.TextCtrl 创建文本框时，可以向其传递以下所示参数。

- parent　　包含该文本框的父组件。
- id　　文本框的 ID。
- value　　文本框中的初始文本。
- pos　　文本框的位置。
- size　　文本框的大小。
- style　　文本框的样式。
- validator　　文本框的验证类。
- name　　文本框的名字。

如果不指明文本框的样式，则将创建一个单行文本框；如果指定其样式为 wx.TE_PASSWORD，则将创建一个密码框。下面所示的 wxPythonTextS.py 脚本创建了一个单行文本框和一个密码框。

```
# -*- coding:utf-8 -*-
# file: wxPythonTextS.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None, title = 'wxPython', size = (300,200))
    # 生成框架窗口
        panel = wx.Panel(frame, -1)                          # 生成面板
        label1 = wx.StaticText(panel, -1, 'wxPython', pos = (120,20)) # 生成标签
        label2 = wx.StaticText(panel, -1, 'User Name:', pos = (10,50))# 生成标签
        text = wx.TextCtrl(panel,                            # 生成文本框
                -1,                                          # 指定文本框 ID
                pos = (100,50),                              # 指定文本框位置
                size = (160, -1))                            # 指定文本框大小
        label3 = wx.StaticText(panel, -1, "Password:",pos = (10,100)) # 生成标签
        password= wx.TextCtrl(panel,                         # 生成文本框
                -1,                                          # 指定文本框 ID
                "password",                                  # 指定初始文本
                pos = (100,100),                             # 指定文本框位置
                size = (160, -1),                            # 指定文本框大小
                style = wx.TE_PASSWORD)                      # 指定文本框为密码框
        frame.Show()
        return True
app = MyApp()
app.MainLoop()
```

运行 wxPythonTextS.py 脚本后，将创建如图 13-8 所示的窗口。



图 13-8　单行文本框和密码框

2. 多行文本框

在创建文本框时，如果指定样式为 wx.TE_MULTILINE，则可以创建一个多行文本框；如果指定样式为 wx.TE_RICH，则可以创建一个可以改变字体和文本颜色的文本框。下面所示的 wxPythonTextM.py 脚本创建了两个多行文本框。

```
# -*- coding:utf-8 -*-
# file: wxPythonTextM.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None,title = 'wxPython',size = (600,400))
        # 生成框架窗口
        panel = wx.Panel(frame, -1)                                    # 生成面板
        label1 = wx.StaticText(panel, -1, 'MultiLine', pos = (280,10))# 生成标签
        text1 = wx.TextCtrl(panel,                                     # 生成文本框
                -1,                                                    # 指定文本框 ID
                pos = (10,30),                                         # 指定文本框位置
                size = (580, 150),                                     # 指定文本框大小
                style=wx.TE_MULTILINE)                                 # 指定文本框样式
        label2 = wx.StaticText(panel, -1, 'RichText', pos = (280,190))# 生成标签
        text2 = wx.TextCtrl(panel,                                     # 生成文本框
                -1,                                                    # 指定文本框 ID
                'Python wxPython',                                     # 指定初始文本
                pos = (10,210),                                        # 指定文本框位置
                size = (580, 150),                                     # 指定文本框大小
                style =wx.TE_MULTILINE|wx.TE_RICH)                     # 指定文本框样式
        text2.SetStyle(0, 6, wx.TextAttr('red', 'blue'))               # 指定文本样式
        frame.Show()
        return True
app = MyApp()
app.MainLoop()
```

运行 wxPythonTextM.py 脚本后，将创建如图 13-9 所示的窗口。



图 13-9　多行文本框

13.2.5 单选框和复选框

在 wxPython 中使用 wx.RadioButton 可以创建单选框，使用 wx.CheckBox 可以创建复选框。使用 wx.RadioButton 创建单选框时可以向其传递以下所示参数。

- parent　　单选框所在的父组件。
- id　　单选框的 ID。
- label　　单选框所显示的文本。
- pos　　单选框的位置。
- size　　单选框的大小。
- style　　单选框的样式。
- validator　　单选框的验证类。
- name　　单选框的名字。

使用 wx.CheckBox 创建复选框时可以向其传递以下所示参数。

- parent　　复选框所在的父组件
- id　　复选框的 ID。
- label　　复选框所显示的文本。
- pos　　复选框的位置。
- size　　复选框的大小。
- style　　复选框的样式。
- name　　复选框的名字。

当创建单选框后，可以使用其 GetValue 方法判断单选框是否被选中，如果被选中则 GetValue 返回真，否则返回假。同样，使用复选框的 IsChecked 方法可以判断复选框是否被选中，如果选中，则 IsChecked 返回真，否则返回假。如下面所示的 wxPythonCheckRadio.py 脚本创建了单选框和复选框。

```
# -*- coding:utf-8 -*-
# file: wxPythonCheckRadio.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None,title = 'wxPython',size = (300,200))
    # 生成框架窗口
        panel = wx.Panel(frame, -1)                          # 生成面板
        self.radio1 = wx.RadioButton(panel,                  # 生成单选框
                -1,                                          # 指定单选框 ID
                'Radio1',                                    # 指定单选框文本
                pos=(10, 40),                                # 指定单选框位置
                style=wx.RB_GROUP)                           # 指定单选框样式
        self.radio2 = wx.RadioButton(panel,                  # 生成框架窗口
                -1,                                          # 指定单选框 ID
```

```
                'Radio2',                                    # 指定单选框文本
                pos=(10, 80))                                # 指定单选框位置
        self.radio3 = wx.RadioButton(panel,                  # 生成单选框
                -1,                                          # 指定单选框 ID
                'Radio3',                                    # 指定单选框文本
                pos=(10, 120))                               # 指定单选框位置
        self.check = wx.CheckBox(panel,                      # 生成复选框
                -1,                                          # 指定复选框 ID
                'CheckBox',                                  # 指定复选框文本
                pos = (120, 40),                             # 指定复选框位置
                size = (150, 20))                            # 指定复选框大小
        self.button1 = wx.Button(panel,-1,'Radio',pos = (120,80)) # 生成按钮
        self.button2 = wx.Button(panel,-1,'Check',pos = (120,120))
        self.Bind(wx.EVT_BUTTON, self.OnButton1, self.button1) # 绑定按钮事件
        self.Bind(wx.EVT_BUTTON, self.OnButton2, self.button2)
        frame.Show()
        return True
    def OnButton1(self, event):                              # 按钮事件处理方法
        if self.radio1.GetValue():                           # 判断 Radio1 是否选中
            self.button1.SetLabel('Radio1')
        elif self.radio2.GetValue():                         # 判断 Radio2 是否选中
            self.button1.SetLabel('Radio2')
        else:
            self.button1.SetLabel('Radio3')
    def OnButton2(self, event):                              # 按钮事件处理方法
        if self.check.IsChecked():                           # 判断 CheckBox 是否选中
            self.button2.SetLabel('Checked')
        else:
            self.button2.SetLabel('UnChecke')
app = MyApp()
app.MainLoop()
```

运行 wxPythonCheckRadio.py 脚本后，将创建如图 13-10 所示的窗口。



图 13-10　单选框和复选框

13.2.6　使用 sizer 布置组件

在 wxPython 中，可以使用 sizer 管理组件的布局。使用 sizer 可以将窗口或其他组件“容器”划分成单元格的形式，然后将组件安排在这些单元格中。这样就不必设置组件的位置，仅仅通过使用 sizer 就可以使组件按所需要的形式进行布局。

wxPython 中有五种形式的 sizer，如表 13-1 所示。

表 13-1 wxPython 中的 sizer

类名	描述
GridSizer	创建基本的网格型 sizer
wx.FlexGridSizer	创建可以根据组件大小改变的 sizer
GridBagSizer	创建可以任意布置组件的 sizer
BoxSizer	创建水平或垂直方向上单行或单列的 sizer
StaticBoxSizer	创建带有边框和标题的 sizer

下面所示的 wxPythonSizer.py 脚本是以创建基本的网格型 sizer 为例，在面板中布置组件。

```
# -*- coding:utf-8 -*-
# file: wxPythonSizer.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None,title = 'wxPython',size = (300,200))
    # 生成框架窗口
        panel = wx.Panel(frame, -1)                          # 生成面板
        sizer = wx.GridSizer(rows=3, cols=3,vgap=0,hgap=0) # 创建一个三行三列的sizer
        sizer.AddSpacer(0)                                   # 向 sizer 中添加一个空项
        label = wx.StaticText(panel, -1, 'label')            # 生成标签
        sizer.Add(label,flag = wx.ALIGN_CENTER)              # 向 sizer 添加标签居中对齐
        sizer.AddSpacer(0)
        button1 = wx.Button(panel, -1, 'Button1')            # 生成按钮
        sizer.Add(button1,flag = wx.ALIGN_CENTER)            # 向 sizer 中添加按钮
        sizer.AddSpacer(0)
        button2 = wx.Button(panel, -1, 'Button2')            # 生成按钮
        sizer.Add(button2,flag = wx.ALIGN_CENTER)            # 向 sizer 中添加按钮
        sizer.AddSpacer(0)
        text = wx.TextCtrl(panel, -1, size = (100,20)) # 生成文本框
        sizer.Add(text)                                      # 向 sizer 中添加文本框
        sizer.AddSpacer(0)
        panel.SetSizer(sizer)                                # 向面板中添加 sizer
        frame.Show()
        return True
app = MyApp()
app.MainLoop()
```

运行 wxPythonSizer.py 脚本后，将创建如图 13-11 所示的窗口。



图 13-11 使用 sizer 布置组件

13.3 对话框

wxPython 提供了消息框和标准的对话框，除此之外，wxPython 还提供了对话框类，通过继承 wxPython 的对话框类，可以创建自定义的对话框，实现与用户交互的功能。

13.3.1 消息框和标准对话框

使用 wxPython 提供的消息框和标准对话框类，可以方便地创建消息框和标准的对话框。其中，标准对话框包含了简单的输入框、文件对话框等。

1. 消息框

使用 wx.MessageBox 可以创建消息框。通过组合其样式可以创建不同形式的消息框。其原型如下。

```
MessageBox(message, caption, style, parent, x, y)
```

其参数含义如下。

- message　　消息框所显示的消息。
- caption　　消息框的标题。
- style　　消息框的样式。
- parent　　消息框的父窗口。
- x　　消息框位置的 X 坐标。
- y　　消息框位置的 Y 坐标。

下面所示的 wxPythonMessageBox.py 脚本创建了几种不同样式的消息框。

```
# -*- coding:utf-8 -*-
# file: wxPythonMessageBox.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        self.frame = wx.Frame(parent = None,title = 'wxPython',size = (300,200))
        # 生成框架窗口
        panel = wx.Panel(self.frame, -1)                                    # 生成面板
        self.button1 = wx.Button(panel,-1,'Style1',pos = (100,20)) # 生成按钮
        self.button2 = wx.Button(panel,-1,'Style2',pos = (100,50))
        self.button3 = wx.Button(panel,-1,'Style3',pos = (100,80))
        self.button4 = wx.Button(panel,-1,'Style4',pos = (100,110))
        self.button5 = wx.Button(panel,-1,'Style5',pos = (100,140))
        self.Bind(wx.EVT_BUTTON, self.OnButton1, self.button1)     # 绑定按钮事件
        self.Bind(wx.EVT_BUTTON, self.OnButton2, self.button2)
        self.Bind(wx.EVT_BUTTON, self.OnButton3, self.button3)
        self.Bind(wx.EVT_BUTTON, self.OnButton4, self.button4)
        self.Bind(wx.EVT_BUTTON, self.OnButton5, self.button5)
        self.frame.Show()
        return True
    def OnButton1(self, event):                                  # 按钮事件处理方法
```

```
        wx.MessageBox('Style1', 'wxPython',                          # 创建 MessageBox
            wx.YES_NO | wx.ICON_QUESTION)
    def OnButton2(self, event):                                      # 按钮事件处理方法
        wx.MessageBox('Sytle2','wxPython',                           # 创建 MessageBox
            wx.OK|wx.CANCEL|wx.ICON_ERROR)
    def OnButton3(self, event):                                      # 按钮事件处理方法
        wx.MessageBox('Style3', 'wxPython',                          # 创建 MessageBox
            wx.OK|wx.CANCEL | wx.ICON_EXCLAMATION)
    def OnButton4(self, event):                                      # 按钮事件处理方法
        wx.MessageBox('Style4', 'wxPython',                          # 创建 MessageBox
            wx.YES_NO|wx.NO_DEFAULT | wx.ICON_HAND)
    def OnButton5(self, event):                                      # 按钮事件处理方法
        wx.MessageBox('Style5', 'wxPython',                          # 创建 MessageBox
            wx.YES_NO|wx.YES_DEFAULT | wx.ICON_INFORMATION)
app = MyApp()
app.MainLoop()
```

运行 wxPythonMessageBox.py 脚本后，将显示如图 13-12 所示窗口，依次单击窗口中的按钮，将分别创建如图 13-13、图 13-14、图 13-15、图 13-16 和图 13-17 所示的几种消息框。



图 13-12 窗口



图 13-13 Style1 消息框



图 13-14 Style2 消息框



图 13-15 Style3 消息框



图 13-16 Style4 消息框



图 13-17 Style5 消息框

2. 标准对话框

wxPython 提供了基本输入对话框以及一些其他标准对话框，其类型如下所示。

- wx.GetTextFromUser() 创建文本输入对话框。
- wx.GetPasswordFromUser() 创建密码输入对话框。
- wx.GetNumberFromUser() 创建整数输入对话框。
- wx.FileDialog() 创建文件打开、关闭对话框。
- wx.FontDialog() 创建选择字体对话框。
- wx.ColourDialog() 创建颜色选择对话框。

其中，wxPython 所创建的标准对话框是 Windows 下标准的对话框样式，与第 12 章中使用 Tkinter 所创建的标准对话框一样。下面所示的 wxPythonStandardDialo.py 脚本仅创建了 2 种基本输入对话框。

```
# -*- coding:utf-8 -*-
# file:  wxPythonStandardDialo.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        self.frame = wx.Frame(parent = None,title = 'wxPython',size = (300,150))
        # 生成框架窗口
        panel = wx.Panel(self.frame, -1)                          # 生成面板
        self.button1 = wx.Button(panel,-1,'Input String',pos = (100,20))
                                                                  # 生成按钮
        self.button2 = wx.Button(panel,-1,'Input Password',pos = (100,70))
        self.Bind(wx.EVT_BUTTON, self.OnButton1, self.button1) # 绑定按钮事件
        self.Bind(wx.EVT_BUTTON, self.OnButton2, self.button2)
        self.frame.Show()
        return True
    def OnButton1(self, event):
        r = wx.GetTextFromUser('wxPython','String', 'Default') # 创建文本输入框
        wx.MessageBox(r, 'wxPython',wx.OK)                        # 创建 MessageBox
    def OnButton2(self, event):
        r = wx.GetPasswordFromUser('wxPython','Password')         # 创建密码输入框
        wx.MessageBox(r, 'wxPython',wx.OK)                        # 创建 MessageBox
app = MyApp()
app.MainLoop()
```

运行 wxPythonStandardDialo.py 脚本后，将显示如图 13-18 所示的窗口，单击窗口中的按钮，将创建如图 13-19 和图 13-20 所示的基本输入对话框。

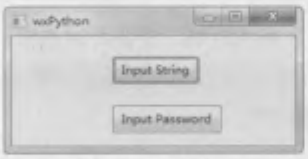


图 13-18　窗口

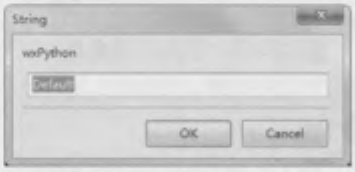


图 13-19　文本输入对话框



图 13-20　密码输入对话框

13.3.2　创建自定义对话框

在 wxPython 中提供了一个名为 wx.Dialog 的对话框类，通过继承 wx.Dialog 可以创建自定义对话框，其所创建的对话框可以像创建的框架窗口一样向其中添加其他组件，并绑定组件事件。

在创建自定义对话框类时，需要在初始化方法中调用 wx.Dialog 的初始化方法。下面所示的 wxPythonDialog.py 脚本是使用 wx.Dialog 创建的自定义对话框。

```
# -*- coding:utf-8 -*-
# file: wxPythonDialog.py
#
import wx                                                    # 导入 wxPython
class MyApp(wx.App):                                         # 通过继承创建类
    def OnInit(self):                                        # 重载 OnInit 方法
        frame = wx.Frame(parent = None,title = 'wxPython',size = (300,170))
                                                             # 生成框架窗口
        panel = wx.Panel(frame, -1)                          # 生成面板
        self.button = wx.Button(panel, -1, 'Show Dialog', pos=(100, 50))
                                                             # 添加 Button
        self.Bind(wx.EVT_BUTTON, self.OnButton, self.button)
        frame.Show()                                         # 显示窗口
        return True
    def OnButton(self, event):                               # 按钮事件响应函数
        dialog = MyDialog()
        r = dialog.ShowModal()                               # 获取返回值
        if r == wx.ID_OK:                                    # 判断是否单击 Ok
            wx.MessageBox('You input:' + dialog.text.GetValue(),# 弹出消息框
                    'wxPython', wx.OK)
        dialog.Destroy()                                     # 销毁对话框
class MyDialog(wx.Dialog):                                   # 定义对话框类
    def __init__(self):                                      # 初始化
        wx.Dialog.__init__(self, None, -1, 'wxDilog',size=(300, 170))
                                                             # 调用父类初始化方法
        label = wx.StaticText(self, -1, 'Simple Dialog',pos = (120,20))# 生成标签
        self.text = wx.TextCtrl(self, -1, pos = (100,50), size = (160, -1))
                                                             # 生成文本框
        self.ok = wx.Button(self, wx.ID_OK, "OK", pos=(50, 80))# 生成 OK 按钮
        self.cancel = wx.Button(self, wx.ID_CANCEL, "Cancel",pos=(200, 80))
                                                             # 生成 Cancel 按钮
app = MyApp()
app.MainLoop()
```

运行 wxPythonDialog.py 脚本后，将显示如图 13-21 所示窗口，单击【Show Dialog】按钮，将创建如图 13-22 所示对话框。



图 13-21 自定义对话框 1



图 13-22 自定义对话框 2

13.4 菜单

在 wxPython 中，创建菜单的过程与使用 PythonWin 中的 MFC 创建菜单过程一样，并且还要

更简单一些。另外，在 wxPython 中，还可以创建右键菜单等弹出式菜单。

13.4.1 创建菜单

在 wxPython 中，创建菜单可以使用 wx.MenuBar 和 wx.Menu 类。其中，wx.MenuBar 创建的是下拉菜单，而 wx.Menu 创建的则是下拉菜单中的菜单命令。

1. 普通菜单

创建普通菜单时，应首先使用 wx.MenuBar 创建下拉菜单的菜单条，然后使用 wx.Menu 创建菜单命令，当菜单命令创建后，可以调用其 Append 方法添加命令，最后调用下拉菜单的 Append 方法将菜单命令添加到下拉菜单中。下面所示的 wxPythonMenu.py 脚本创建了一组菜单。

```
# -*- coding:utf-8 -*-
# file: wxPythonMenu.py
#
import wx                                                    # 导入 wxPython
class MyApp(wx.App):                                         # 通过继承创建类
    def OnInit(self):                                        # 重载 OnInit 方法
        frame = wx.Frame(parent = None,title = 'wxPython',size = (300,170))
                                                             # 生成框架窗口
        panel = wx.Panel(frame, -1)                          # 生成面板
        menuBar = wx.MenuBar()                               # 创建菜单条
        menu = wx.Menu()                                     # 创建菜单
        open = menu.Append(-1, 'Open')                       # 向菜单中添加 Open
        exit = menu.Append(-1, 'Save')                       # 向菜单中添加 Save
        menu.AppendSeparator()                               # 向菜单中添加分隔符
        close = menu.Append(-1, 'Close')                     # 向菜单中添加 Close
        menuBar.Append(menu, '&File')                        # 向菜单条中添加 File
        menu = wx.Menu()                                     # 重新创建菜单
        copy = menu.Append(-1, 'Copy')                       # 向菜单中添加 Copy
        paste = menu.Append(-1, 'Paste')                     # 向菜单中添加 Paste
        cut = menu.Append(-1, 'Cut')                         # 向菜单中添加 Cut
        menu.AppendSeparator()                               # 向菜单中添加分隔符
        selectall = menu.Append(-1, 'SelectAll')             # 向菜单中添加 SelectAll
        menuBar.Append(menu, '&Edit')                        # 向菜单条中添加 Edit
        menu = wx.Menu()                                     # 重新创建菜单
        about = menu.Append(-1, 'About')                     # 向菜单中添加 About
        menuBar.Append(menu, '&Help')                        # 向菜单条中添加 Help
        frame.SetMenuBar(menuBar)                            # 向框架窗口中添加菜单
        frame.Show()                                         # 显示窗口
        return True
app = MyApp()
app.MainLoop()
```

运行 wxPythonMenu.py 脚本后，将创建如图 13-23、图 13-24 和图 13-25 所示的菜单。

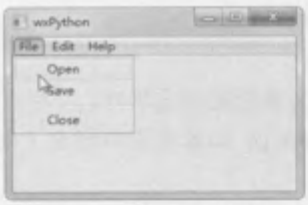


图 13-23 File 菜单

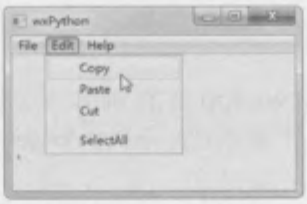


图 13-24 Edit 菜单



图 13-25 Help 菜单

2. 弹出式菜单

要想创建弹出式菜单，则需要使用 wx.Menu 类来创建，并使用 PopupMenu 方法显示菜单。下面所示的 wxPythonPopupMenu.py 脚本绑定了鼠标右击事件，当右击鼠标时使用 event 对象的 GetX 和 GetY 方法获取鼠标位置，然后使用 PopupMenu 方法显示弹出式菜单。

```
# -*- coding:utf-8 -*-
# file: wxPythonPopupMenu.py
#
import wx                                                    # 导入 wxPython
class MyApp(wx.App):                                         # 通过继承创建类
    def OnInit(self):                                        # 重载 OnInit 方法
        frame = wx.Frame(parent = None,title = 'wxPython',size = (300,170))
    # 生成框架窗口
        self.panel = wx.Panel(frame, -1)                     # 生成面板
        menuBar = wx.MenuBar()                               # 创建菜单条
        self.menu = wx.Menu()                                # 创建菜单
        open = self.menu.Append(-1, 'Open')
        save = self.menu.Append(-1, 'Save')
        self.menu.AppendSeparator()
        close = self.menu.Append(-1, 'Close')
        menuBar.Append(self.menu, '&File')
        frame.SetMenuBar(menuBar)                            # 向框架窗口中添加菜单
        self.Bind(wx.EVT_RIGHT_DOWN, self.OnRClick)          # 绑定右键事件
        frame.Show()
        return True
    def OnRClick(self, event):
        pos = (event.GetX(),event.GetY())                    # 获得鼠标单击坐标
        self.panel.PopupMenu(self.menu, pos)                 # 显示菜单
app = MyApp()
app.MainLoop()
```

运行 wxPythonPopupMenu.py 脚本后，用鼠标右击窗口空白处，将弹出如图 13-26 所示的菜单。

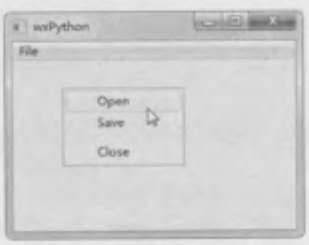


图 13-26 右键弹出式菜单

13.4.2 绑定菜单事件

同绑定按钮事件一样，只要使用 wx.App 类的 Bind 方法即可为菜单项绑定事件。不同的是，菜单的绑定事件为 wx.EVT_MENU。下面所示的 wxPythonMenuEvent.py 脚本将菜单绑定了相应的事件处理函数。

```
# -*- coding:utf-8 -*-
# file: wxPythonMenuEvent.py
#
import wx                                                    # 导入 wxPython
class MyApp(wx.App):                                         # 通过继承创建类
    def OnInit(self):                                        # 重载 OnInit 方法
        self.frame = wx.Frame(parent = None,title = 'wxPython',size = (300,170))
    # 生成框架窗口
        self.panel = wx.Panel(self.frame, -1)                # 生成面板
        menuBar = wx.MenuBar()                               # 创建菜单条
        self.menu = wx.Menu()                                # 创建菜单
        open = self.menu.Append(-1, 'Open')
        save = self.menu.Append(-1, 'Save')
        self.menu.AppendSeparator()
        close = self.menu.Append(-1, 'Close')
        menuBar.Append(self.menu, '&File')
        self.menu = wx.Menu()                                # 重新创建菜单
        about = self.menu.Append(-1, 'About')
        menuBar.Append(self.menu, '&Help')
        self.frame.SetMenuBar(menuBar)                       # 向框架窗口中添加菜单
        self.Bind(wx.EVT_MENU, self.OnOpen, open)            # 绑定菜单事件
        self.Bind(wx.EVT_MENU, self.OnSave, save)
        self.Bind(wx.EVT_MENU, self.OnClose, close)
        self.Bind(wx.EVT_MENU, self.OnAbout, about)
        self.Bind(wx.EVT_RIGHT_DOWN, self.OnRClick)
        self.frame.Show()
        return True
    def OnOpen(self, event):                                 # 处理 Open 命令
        dialog = wx.FileDialog(None, 'wxPython', style = wx.FD_OPEN)
                                                             # 创建打开文件对话框
        dialog.ShowModal()
        dialog.Destroy()
    def OnSave(self, event):                                 # 处理 Save 命令
        dialog = wx.FileDialog(None, 'wxPython', style =  wx.FD_SAVE)
                                                             # 创建保存文件对话框
        dialog.ShowModal()
        dialog.Destroy()
    def OnClose(self, event):                                # 处理 Close 命令
        self.frame.Destroy()                                 # 退出程序
    def OnAbout(self, event):                                # 处理 About 命令
        wx.MessageBox('wxPython Menu Event', 'wxPython', wx.OK)    # 创建消息框
    def OnRClick(self, event):                               # 处理右键事件
        pos = (event.GetX(),event.GetY())
        self.panel.PopupMenu(self.menu, pos)                 # 创建弹出式菜单
app = MyApp()
app.MainLoop()
```

13.5 一个简单的文本编辑器

wxPython 提供的文本框组件功能十分强大，通过使用 wxPython 提供的文本框组件即可创建一个简单的文本编辑器。

下面所示的 wxEditor.py 脚本是使用 wxPython 所提供的文本框组件实现了打开文件、关闭文件、粘贴文本、复制文本等操作。另外，还可以改变文本框的背景色和前景色，以及修改窗口的透明度等。

```
# -*- coding:utf-8 -*-
# file: wxEditor.py
#
import wx                                                    # 导入 wxPython
class CreateMenu():                                          # 创建菜单类
    def __init__(self, parent):
        self.menuBar = wx.MenuBar()                          # 创建菜单条
        self.file = wx.Menu()                                # 创建菜单
        self.open = self.file.Append(-1, '打开')
        self.save = self.file.Append(-1, '保存')
        self.saveas = self.file.Append(-1, '另存为')
        self.file.AppendSeparator()
        self.close = self.file.Append(-1, '退出')
        self.menuBar.Append(self.file, '文件(&F)')
        self.edit = wx.Menu()
        self.undo = self.edit.Append(-1, '撤销')
        self.redo = self.edit.Append(-1, '重做')
        self.edit.AppendSeparator()
        self.cut = self.edit.Append(-1, '剪切')
        self.copy = self.edit.Append(-1, '复制')
        self.paste = self.edit.Append(-1, '粘贴')
        self.edit.AppendSeparator()
        self.selectall = self.edit.Append(-1, '全选')
        self.menuBar.Append(self.edit,'编辑(&E)')
        self.view = wx.Menu()
        self.color = self.view.AppendCheckItem(1051, '设为黑色')
        self.trans = self.view.Append(-1, '设置透明度')
        self.menuBar.Append(self.view, '查看(&V)')
        self.help = wx.Menu()
        self.about = self.help.Append(-1, '关于')
        self.menuBar.Append(self.help, '帮助(&H)')
        parent.SetMenuBar(self.menuBar)                      # 向框架窗口中添加菜单
class MyApp(wx.App):                                         # 通过继承创建类
    def OnInit(self):                                        # 重载 OnInit 方法
        self.file = ''
        self.width = 600
        self.height = 480
        self.frame = wx.Frame(parent = None,title = 'wxPython Notebook',
                size = (self.width,self.height))             # 生成框架窗口
        self.panel = wx.Panel(self.frame, -1)                # 生成面板
        self.menu = CreateMenu(self.frame)                   # 生成菜单
```

```
        self.text = wx.TextCtrl(self.panel,                    # 生成文本框
                -1,
                pos = (2,2),
                size = (self.width-10, self.height-50),
                style = wx.HSCROLL | wx.TE_MULTILINE)
        self.Bind(wx.EVT_MENU, self.OnOpen, self.menu.open)# 绑定事件
        self.Bind(wx.EVT_MENU, self.OnSave, self.menu.save)
        self.Bind(wx.EVT_MENU, self.OnSaveAs, self.menu.saveas)
        self.Bind(wx.EVT_MENU, self.OnClose, self.menu.close)
        self.Bind(wx.EVT_MENU, self.OnUndo, self.menu.undo)
        self.Bind(wx.EVT_MENU, self.OnRedo, self.menu.redo)
        self.Bind(wx.EVT_MENU, self.OnCut, self.menu.cut)
        self.Bind(wx.EVT_MENU, self.OnCopy, self.menu.copy)
        self.Bind(wx.EVT_MENU, self.OnPaste, self.menu.paste)
        self.Bind(wx.EVT_MENU, self.OnSelectAll, self.menu.selectall)
        self.Bind(wx.EVT_MENU, self.OnColor, self.menu.color)
        self.Bind(wx.EVT_MENU, self.OnTrans, self.menu.trans)
        self.Bind(wx.EVT_MENU, self.OnAbout, self.menu.about)
        self.Bind(wx.EVT_RIGHT_DOWN, self.OnRClick)
        self.Bind(wx.EVT_SIZE, self.Resize)
        self.frame.Show()
        return True
    def OnOpen(self, event):                                   # 处理打开命令
        dialog = wx.FileDialog(None, 'wxPython Notebook', style = wx.FD_OPEN)
        if dialog.ShowModal() == wx.ID_OK:
            self.file = dialog.GetPath()
            file = open(self.file)
            self.text.Clear()
            self.text.WriteText(file.read())
            file.close()
        dialog.Destroy()
    def OnSave(self, event):                                   # 处理保存命令
        if self.file == '':
            dialog = wx.FileDialog(None, 'wxPython Notebook', style =  wx.FD_SAVE)
            if dialog.ShowModal() == wx.ID_OK:
                self.file = dialog.GetPath()
                self.text.SaveFile(self.file)
            dialog.Destroy()
        else:
            self.text.SaveFile(self.file)
    def OnSaveAs(self, event):                                 # 处理另存为命令
        dialog = wx.FileDialog(None, 'wxPython Notebook', style =  wx.FD_SAVE)
        if dialog.ShowModal() == wx.ID_OK:
            self.file = dialog.GetPath()
            self.text.SaveFile(self.file)
        dialog.Destroy()
    def OnClose(self, event):                                  # 处理退出命令
        self.frame.Destroy()
    def OnAbout(self, event):                                  # 处理关于命令
        wx.MessageBox('A simple editor!', 'wxPython Notebook', wx.OK)
    def OnRClick(self, event):                                 # 处理右键事件
        pos = (event.GetX(),event.GetY())
        self.panel.PopupMenu(self.menu.edit, pos)
    def OnUndo(self, event):                                   # 处理撤销命令
        self.text.Undo()
    def OnRedo(self, event):                                   # 处理重做命令
        self.text.Redo()
```

```
    def OnCut(self, event):                                    # 处理剪切命令
        self.text.Cut()
    def OnCopy(self, event):                                   # 处理复制命令
        self.text.Copy()
    def OnPaste(self, event):                                  # 处理粘贴命令
        self.text.Paste()
    def OnSelectAll(self, event):                              # 处理全选命令
        self.text.SelectAll()
    def OnColor(self, event):                                  # 处理设为黑色命令
        if self.menu.view.IsChecked(1051):
            self.text.SetBackgroundColour('black')
            self.text.SetForegroundColour('green')
            self.text.Refresh()
        else:
            self.text.SetBackgroundColour('white')
            self.text.SetForegroundColour('black')
            self.text.Refresh()
    def OnTrans(self, event):                                  # 处理设置透明度命令
        r = wx.GetNumberFromUser('请选择透明度','',
                'wxPython Notebook',80, min = 30)
        if r != -1:
            self.frame.SetTransparent((r* 255/100))
            self.frame.Refresh()
    def Resize(self, event):                                   # 处理窗口改变大小命令
        newsize = self.frame.GetSize()
        width = newsize.GetWidth() - 10
        height = newsize.GetHeight() - 50
        self.text.SetSize((width,height))
        self.text.Refresh()
app = MyApp()
app.MainLoop()
```

运行 wxEditor.py 后，单击菜单【文件】|【打开】命令，打开【wxEditor_ansi.py】文件，如图 13-27 所示。单击菜单【查看】|【设为黑色】命令，可以将背景设为黑色，文字设为绿色，如图 13-28 所示。



图 13-27 简单的文本编辑器 wxEditor



图 13-28 将 wxEditor 背景设为黑色，文字设为绿色

需要注意的是，这个文本编辑器只能读取 ANSI 字符集文件，如果是使用 UTF-8 编码保存的文件，则无法打开。

13.6 本章小结

本章介绍了另一种 GUI 设计模块：wxPython。因为其不是 Python 的内置模块，因此本章首先介绍了 wxPython 的下载和安装，接着介绍了创建 wxPython 各种组件的方法，这些组件包括面板、按钮、标签、文本框、单选框和复选框等。然后介绍了使用 wxPython 消息框和标准对话框的方法，还介绍了创建菜单、绑定菜单事件的方法。最后，演示了使用 wxPython 设计一个简单文本编辑器的案例。

下一章将介绍如何使用 PyGTK 模块设计 GUI 程序。

◆ ◆ ◆

第 14 章　使用 PyGTK 编写 GUI

本章包括

- PyGTK 概述
- 使用 PyGTK 消息框
- 创建自定义对话框
- 响应菜单事件
- 使用 PyGTK 组件
- 使用 PyGTK 标准对话框
- 创建菜单
- 创建和使用资源文件

GTK 是开源的图形用户界面库，虽然 GTK 库是使用 C 语言编写的，但其使用了类的思想。GTK 库可以运行在包括 Windows 在内的多种操作系统上。PyGTK 是对 GTK 的封装，通过使用 PyGTK 模块，可以在 Python 中使用 GTK 来创建 GUI 界面。

遗憾的是，目前，PyGTK 运行库对 Python 版本的支持最高还只到 Python 2.7，因此，本章将以 Python 2.7 为基础，介绍 PyGTK 运行库的使用，了解 PyGTK 运行库的基本使用方法。当 PyGTK 运行库推出支持 Python 3 的版本之后，就可以很容易地上手了。

14.1　PyGTK 概述

由于 GTK 主要应用在 Linux 平台下，因此其风格与 Windows 的风格有所不同。在 Windows 下使用 PyGTK 时，需要使用 GTK 的运行库，可以到 PyGTK 官方提供的安装程序中下载 GTK 的运行库。

14.1.1　PyGTK 安装

由于 PyGTK 不是 Python 官方安装程序的一部分，因此要从其官方网站 http://www.pygtk.org/ 下载 PyGTK 的安装程序。在 Windows 下安装 PyGTK 时，需要安装运行 PyGTK 所需要的动态链接库文件。为了简便起见，推荐下载官方的 “PyGTK all-in-one installer for win32”，该安装程序将安装在 Windows 下使用 PyGTK 所需的全部文件，并且设置系统环境变量。

PyGTK 的安装步骤如下。

step 1　下载 PyGTK 的安装程序 pygtk-all-in-one-2.24.2.win32-py2.7.msi，直接双击安装程序进行安装，将显示如图 14-1 所示对话框。



在安装之前应先将 Python 2.7 安装好。

step 2　单击【Next】按钮，进入下一个界面，如图 14-2 所示。此时可以选择需要安装的 PyGTK 运行库的安装包。



图 14-1　PyGTK 安装程序



图 14-2　选择安装内容

step 3　单击【Next】按钮，进入下一个界面，如图 14-3 所示，这时可直接单击【Install】按钮，开始安装 PyGTK 运行库。

step 4　经过一段时间的安装后，将出现如图 14-4 所示的界面，表示安装完成，单击【Finish】按钮完成安装。



图 14-3　开始安装



图 14-4　安装完成

由于安装程序设置了系统的环境变量，因此安装完成后需要重新启动系统，使环境变量生效。重启系统后可以在 Python 交互式 shell 中输入以下语句来验证 PyGTK 是否安装成功。

```
import pygtk
import gtk
```

如果上述语句成功运行，则表示 PyGTK 已经安装成功，即可以使用 PyGTK 进行 GUI 编程了。

14.1.2　创建窗口

使用 gtk 模块中的 gtk.Window 类可以创建一个窗口。窗口的属性（如大小、标题等）需要调用窗口对象的相应方法进行设置。当完成窗口设置后，即可调用窗口对象的 show 方法显示窗口，然后调用 gtk.main 函数进入消息循环中。

常用的窗口属性设置方法如下。

◆ set_title(title)　　设置窗口标题。

- set_position(position)　　设置窗口位置。
- fullscreen()　　设置为全屏窗口。
- set_default_size(width, height)　　设置窗口的默认大小。

下面所示的 HelloGTK.py 脚本是使用 gtk.Window()创建了一个 GUI 窗口。

```
# -*- coding:utf-8 -*-
# file: HelloGTK.py
#
import pygtk                                          # 导入 pygtk 模块
pygtk.require('2.0')                                  # 设置 pygtk 所需的 gtk 版本
import gtk                                            # 导入 gtk 模块
window = gtk.Window()                                 # 创建窗口对象
window.set_title('PyGTK')                             # 设置窗口标题
window.set_default_size(300, 200)                     # 设置窗口大小
window.show()                                         # 显示窗口
gtk.main()                                            # 进入消息循环
```

运行 HelloGTK.py 脚本，将创建如图 14-5 所示的窗口。



图 14-5　PyGTK 窗口

虽然 PyGTK 是对 C 语言库 GTK 的封装，但由于 GTK 在设计时就使用了类的思想，因此在使用 PyGTK 时最好使用类的方式。下面所示的 HelloGTK++.py 改写了上例中的 HelloGTK.py 脚本。

```
# -*- coding:utf-8 -*-
# file: HelloGTK++.py
#
import pygtk                                                  # 导入 pygtk 模块
pygtk.require('2.0')                                          # 设置 pygtk 所需的 gtk 版本
import gtk                                                    # 导入 gtk 模块
class MyWindow():                                             # 定义窗口类
    def __init__(self, title, width, height):                 # 定义初始化方法
        self.window = gtk.Window()                            # 生成窗口对象
        self.window.set_title(title)                          # 设置窗口标题
        self.window.set_default_size(width, height)           # 设置窗口大小
        self.window.show()                                    # 显示窗口
    def main(self):                                           # 定义 main 方法
        gtk.main()                                            # 调用 gtk.main 方法
window = MyWindow('PyGTK', 300, 200)                          # 创建窗口对象
window.main()                                                 # 进入消息循环
```

14.2 组件

PyGTK 提供了许多常用组件，这些组件的风格可能与 Windows 中常见的组件风格有所不同。在 PyGTK 中创建组件略微烦琐，组件创建后往往需要调用组件对象的方法才能设置组件的属性。本节将介绍 PyGTK 中常用组件的使用。

14.2.1 标签

PyGTK 中的标签用于显示文本。当创建标签以后，可以调用标签对象的方法设置标签中的文字、文字的对齐方式、标签的角度等。

1. 创建标签

使用 gtk.Label 可以创建标签并设置标签的文本。标签对象有以下几种方法用于设置，或者获取标签的属性。

- set_text()　　重新设置标签文本。
- get_text()　　获取标签文本。
- set_justify()　　设置标签中文本的对齐方式。
- get_justify()　　获取标签中文本的对齐方式。
- set_angle()　　设置标签的角度，角度值应为 90 或 270。
- get_angle()　　获取标签的角度。

下面所示的 PyGTKLabel.py 脚本是使用 gtk.Label 创建标签。

```
# -*- coding:utf-8 -*-
# file: PyGTKLabel.py
#
import pygtk                                              # 导入 pygtk 模块
pygtk.require('2.0')                                      # 设置 pygtk 所需的 gtk 版本
import gtk                                                # 导入 gtk 模块
class MyWindow():                                         # 定义窗口类
    def __init__(self, title, width, height):             # 定义初始化方法
        self.window = gtk.Window()                        # 生成窗口对象
        self.window.set_title(title)                      # 设置窗口标题
        self.window.set_default_size(width, height)       # 设置窗口大小
        label = gtk.Label('PyGTK')                        # 创建标签
        label.set_angle(90)                               # 设置标签角度
        self.window.add(label)                            # 向窗口中添加标签
        label.show()                                      # 显示标签
        self.window.show()                                # 显示窗口
    def main(self):                                       # 定义 main 方法
        gtk.main()                                        # 调用 gtk.main 方法
window = MyWindow('PyGTK', 300, 200)                      # 创建窗口对象
window.main()
```

运行 PyGTKLabel.py 脚本后，将创建如图 14-6 所示的窗口，从图中可看出，标签中的文字内容旋转了 90 度。



图 14-6 创建的标签

2. 设置 PyGTK 默认字体

运行上例的 PyGTKLabel.py 脚本后，如果在命令行窗口中出现无法载入字体的警告，则可以通过设置 PyGTK 的默认字体来解决。假设 PyGTK 的安装目录为“C:\Python27\”，则使用文本编辑器打开“C:\Python27\Lib\site-packages\gtk-2.0\runtime\etc\gtk-2.0”目录下的“gtkrc”文件，其内容如下。

```
gtk-theme-name = "MS-Windows"
```

将其修改为如下所示的内容。

```
style "user-font"
{
    font_name="Tahoma,SimSun 10"
}
widget_class "*" style "user-font"
gtk-font-name = "Tahoma, Simsun 10"
gtk-theme-name = "MS-Windows"
```

如果需要在 PyGTK 的组件中使用中文，则除了在脚本的首行使用“# -*- coding:utf-8 -*-”外，还应该将脚本保存成 UTF-8 的编码格式。

3. 向窗口中添加多个标签

由于 gtk.window 对象每次只能容纳一个组件，因此如果需要向其中添加多个组件，则需要使用 Box 对象。PyGTK 中提供了 gtk.HBox()和 gtk.VBox()来创建 Box 对象，它们可以相互嵌套。当创建 Box 对象后，可以使用其 pack_start 和 pack_end 方法向其中添加组件。其原型分别如下。

```
pack_start(child, expand=True, fill=True, padding=0)
pack_end(child, expand=True, fill=True, padding=0)
```

其参数含义如下。

- child　　向 Box 对象中添加的组件。
- expand　　Bool 型，是否允许组件获取更大空间。
- fill　　Bool 型，只有在 expand 为 True 时有效。
- padding　　Box 对象中组件间的间距。

对于 pack_start 方法，其每次添加的组件应位于所有使用 pack_start 方法添加的组件之后。对

于 pack_end 方法，其每次添加的组件应位于所有使用 pack_end 方法添加的组件之前。

下面所示的 PyGTKLabelM.py 脚本是使用 Box 对象向窗口中添加多个标签。

```
# -*- coding:utf-8 -*-
# file: PyGTKLabelM.py
#
import pygtk                                                    # 导入 pygtk 模块
pygtk.require('2.0')                                            # 设置 pygtk 所需的 gtk 版本
import gtk                                                      # 导入 gtk 模块
class MyWindow():                                               # 定义窗口类
    def __init__(self, title, width, height):                   # 定义初始化方法
        self.window = gtk.Window()                              # 生成窗口对象
        self.window.set_title(title)                            # 设置窗口标题
        self.window.set_default_size(width, height)             # 设置窗口大小
        vbox = gtk.VBox(False, 5)                               # 生成竖向 Box 对象
        hbox1 = gtk.HBox(False, 5)                              # 生成水平 Box 对象
        hbox2 = gtk.HBox(False, 5)
        label1 = gtk.Label('Label1')                            # 创建标签
        label1.set_angle(90)                                    # 设置标签角度
        label2 = gtk.Label('Label2')
        label2.set_angle(270)
        label3 = gtk.Label('Label3')
        label4 = gtk.Label('Label4')
        label5 = gtk.Label('Label5')
        hbox1.pack_start(label1)                                # 向 Box 对象中添加标签
        hbox1.pack_start(label2)
        hbox2.pack_start(label3)
        hbox2.pack_end(label4)
        hbox2.pack_end(label5)
        vbox.pack_start(hbox1)                                  # 向 Box 对象中添加其他 Box 对象
        vbox.pack_start(hbox2)
        self.window.add(vbox)                                   # 向窗口添加 Box 对象
        label1.show()                                           # 显示标签
        label2.show()
        label3.show()
        label4.show()
        label5.show()
        hbox1.show()                                            # 显示 Box 对象
        hbox2.show()
        vbox.show()
        self.window.show()                                      # 显示窗口
    def main(self):                                             # 定义 main 方法
        gtk.main()                                              # 调用 gtk.main 方法
window = MyWindow('PyGTK', 300, 200)                            # 创建窗口对象
window.main()
```

运行 PyGTKLabelM.py 脚本后，将创建如图 14-7 所示的窗口。



图 14-7 创建多个标签

14.2.2 按钮

在 PyGTK 中，使用 gtk.Button 可以创建按钮，通过按钮的 connect 方法可以将按钮事件信号绑定到事件处理函数上。

1. 创建按钮

使用 gtk.Button 创建按钮对象后，可以使用以下几个按钮对象的方法设置或者获取按钮的属性。

- pressed() 产生 pressed 信号。
- released() 产生 released 信号。
- clicked() 产生 clicked 信号。
- enter() 产生 enter 信号。
- leave() 产生 leave 信号。
- set_label() 设置按钮文字。
- get_label() 获得按钮文字。

下面所示的 PyGTKButton.py 脚本是通过 HBox 创建了两个水平放置的按钮。

```
# -*- coding:utf-8 -*-
# file: PyGTKButton.py
#
import pygtk                                                    # 导入 pygtk 模块
pygtk.require('2.0')                                            # 设置 pygtk 所需的 gtk 版本
import gtk                                                      # 导入 gtk 模块
class MyWindow():                                               # 定义窗口类
    def __init__(self, title, width, height):                   # 定义初始化方法
        self.window = gtk.Window()                              # 生成窗口对象
        self.window.set_title(title)                            # 设置窗口标题
        self.window.set_default_size(width, height)             # 设置窗口大小
        hbox = gtk.HBox(False, 20)                              # 生成水平 Box 对象
        button1 = gtk.Button('Button1')                         # 创建按钮
        button2 = gtk.Button('Button2')
        hbox.pack_start(button1)                                # 向 Box 对象中添加按钮
        hbox.pack_start(button2)
        self.window.add(hbox)                                   # 向窗口添加 Box 对象
        hbox.show()                                             # 显示 Box 对象
        button1.show()                                          # 显示按钮
```

```
        button2.show()
        self.window.show()                                    # 显示窗口
    def main(self):                                           # 定义 main 方法
        gtk.main()                                            # 调用 gtk.main 方法
window = MyWindow('PyGTK', 150, 30)                           # 创建窗口对象
window.main()
```

运行 PyGTKButton.py 脚本后，将创建如图 14-8 所示的窗口，在这个窗口中有两个水平放置的按钮。

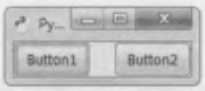


图 14-8　创建的两个水平放置的按钮

2. 按钮事件

为窗口中的按钮绑定按钮事件，可以使用按钮对象的 connect 方法，其原型如下。

```
connect(name, func, func_data)
```

其参数含义如下。

- ◆ name　　事件信号名，按钮可以是 clicked、pressed 等。
- ◆ func　　事件响应函数。
- ◆ func_data　　可选参数，传递给事件相应函数的附加数据。

其中，事件响应函数应该根据 connect 函数所使用的参数来定义。如果在 conncet 函数中使用了 func_data 参数，则事件响应函数应声明成如下形式。

```
def func(self, widget, data):
    <处理语句>
```

如果在 conncet 函数中没有使用 func_data 参数，则事件响应函数应声明成如下所示形式。

```
def func(self, widget):
    <处理语句>
```

下面所示的 PyGTKButtonEvent.py 脚本是使用按钮对象的 connect 方法绑定按钮事件。

```
# -*- coding:utf-8 -*-
# file: PyGTKButtonEvent.py
#
import pygtk                                                  # 导入 pygtk 模块
pygtk.require('2.0')                                          # 设置 pygtk 所需的 gtk 版本
import gtk                                                    # 导入 gtk 模块
class MyWindow():                                             # 定义窗口类
    def __init__(self, title, width, height):                 # 定义初始化方法
        self.window = gtk.Window()                            # 生成窗口对象
        self.window.set_title(title)                          # 设置窗口标题
        self.window.set_default_size(width, height)           # 设置窗口大小
        self.window.connect('destroy', lambda q: gtk.main_quit()) # 关闭窗口时退出程序
```

```
        hbox = gtk.HBox(False, 20)                          # 生成水平 Box 对象
        self.button1 = gtk.Button('Button1')                # 创建按钮
        self.button2 = gtk.Button('Button2')
        self.button1.connect('clicked', self.OnButton1, 'Button1') # 绑定按钮事件
        self.button2.connect('clicked', self.OnButton2, 'Button2')
        hbox.pack_start(self.button1)                       # 向 Box 对象中添加按钮
        hbox.pack_start(self.button2)
        self.window.add(hbox)                               # 向窗口添加 Box 对象
        hbox.show()                                         # 显示 Box 对象
        self.button1.show()                                 # 显示按钮
        self.button2.show()
        self.window.show()                                  # 显示窗口
    def main(self):                                         # 定义 main 方法
        gtk.main()                                          # 调用 gtk.main 方法
    def OnButton1(self,widget, data):                       # 处理按钮事件
        self.button2.set_label('Quit')                      # 重新设置 Button2 文本
    def OnButton2(self, widgte, data):                      # 处理按钮事件
        gtk.main_quit()                                     # 退出程序
window = MyWindow('PyGTK', 150, 30)                         # 创建窗口对象
window.main()
```

运行 PyGTKButtonEvent.py 脚本后，将显示如图 14-9 所示窗口，单击【Button1】按钮，可以将【Button2】的标签为“Quit”，如图 14-10 所示。



图 14-9 窗口初始界面

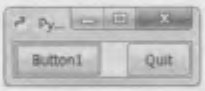


图 14-10 改变按钮标签

单击【Button2】按钮（或右侧显示“Quit”文字的按钮）可以退出窗口。另外，脚本中还使用了 Window 对象的 connect 绑定了“destroy”信号，之前的脚本关闭窗口后，脚本并未退出（在 Windows 命令窗口中需使用 Ctrl+C 退出）。这是因为关闭窗口后仅发送“destroy”信号，但并没有退出 gtk.main()，而在该脚本中使用 lambda 将 Window 对象的“destroy”信号绑定到 gtk.main_quit()函数后，这样当窗口关闭时就可以退出脚本了。

14.2.3 容器组件

使用 Box 对象放置组件时不能调整组件的大小，也不能任意设置组件的坐标位置。对于比较复杂 GUI 需求，使用 Box 对象显然不能满足要求，在 PyGTK 中，还有其他的容器组件用于放置其他组件。例如，Fixed 组件和 Layout 组件，它们都可以使用坐标的形式布置组件。

1. Fixed 组件

使用 gtk.Fixed()可以创建一个 Fixed 组件，而通过 Fixed 组件对象的 put 方法则可以向 Fixed 组件中添加其他组件。而使用 move 方法则可以改变组件在 Fixed 组件中的位置。put 方法和 move 方法的原型分别如下。

```
put(widget, x, y)
move(widget, x, y)
```

这两个方法的参数含义相同，如下所示。

- widget　要添加或者移动的组件对象。
- x　X 坐标值。
- y　Y 坐标值。

下面所示的 PyGTKFixed.py 脚本是使用 Fixed 组件向窗口中添加多个组件。

```
# -*- coding:utf-8 -*-
# file: PyGTKFixed.py
#
import pygtk                                          # 导入 pygtk 模块
pygtk.require('2.0')                                  # 设置 pygtk 所需的 gtk 版本
import gtk                                            # 导入 gtk 模块
class MyWindow():                                     # 定义窗口类
    def __init__(self, title, width, height):         # 定义初始化方法
        self.window = gtk.Window()                    # 生成窗口对象
        self.window.set_title(title)                  # 设置窗口标题
        self.window.set_default_size(width, height)   # 设置窗口大小
        self.window.connect('destroy', lambda q: gtk.main_quit())
        self.fixed = gtk.Fixed()
        self.label = gtk.Label('PyGTK')               # 创建标签
        self.fixed.put(self.label,10,5)               # 添加标签
        self.button = gtk.Button('Move')              # 创建按钮
        self.button.connect('clicked',self.OnButton, 'Move')   # 绑定按钮事件
        self.fixed.put(self.button, 120, 150)         # 添加按钮
        self.window.add(self.fixed)                   # 向窗口中添加 Fixed
        self.label.show()                             # 显示标签
        self.button.show()                            # 显示按钮
        self.fixed.show()                             # 显示 Fixed 组件
        self.window.show()                            # 显示窗口
    def OnButton(self, widget, data):
        self.fixed.move(self.label, 100,50)
    def main(self):                                   # 定义 main 方法
        gtk.main()
window = MyWindow('PyGTK', 300, 200)                 # 创建窗口对象
window.main()
```

运行 PyGTKFixed.py 脚本后将创建如图 14-11 所示的窗口。单击【Move】按钮，标签将移动到另一个位置，如图 14-12 所示。

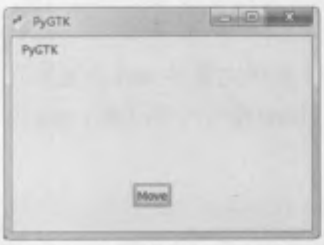


图 14-11　使用 Fixed 组件



图 14-12　移动后的标签位置

2. Layout 组件

Layout 组件和 Fixed 组件基本相同。Layout 组件也具有 put 方法和 move 方法，可以添加或者移动组件的位置。下面所示的 PyGTKLayout.py 脚本是使用 Layout 组件向窗口中添加多个组件。

```
# -*- coding:utf-8 -*-
# file: PyGTKLayout.py
#
import pygtk                                          # 导入 pygtk 模块
pygtk.require('2.0')                                  # 设置 pygtk 所需的 gtk 版本
import gtk                                            # 导入 gtk 模块
class MyWindow():                                     # 定义窗口类
    def __init__(self, title, width, height):         # 定义初始化方法
        self.x = 10                                   # 定义坐标信息
        self.y = 5
        self.window = gtk.Window()                    # 生成窗口对象
        self.window.set_title(title)                  # 设置窗口标题
        self.window.set_default_size(width, height)   # 设置窗口大小
        self.window.connect('destroy', lambda q: gtk.main_quit())
        self.layout = gtk.Layout()
        self.label = gtk.Label('PyGTK')               # 创建标签
        self.layout.put(self.label,self.x,self.y)     # 添加标签
        self.button = gtk.Button('Move')              # 创建按钮
        self.button.connect('clicked',self.OnButton, 'Move')   # 绑定按钮事件
        self.layout.put(self.button, 120, 150)        # 添加按钮
        self.window.add(self.layout)                  # 向窗口中添加 Layout
        self.label.show()                             # 显示标签
        self.button.show()                            # 显示按钮
        self.layout.show()                            # 显示 Layout 组件
        self.window.show()                            # 显示窗口
    def OnButton(self, widget, data):                 # 按钮事件响应函数
        self.x = self.x + 5
        self.y = self.y + 5
        if self.x >= 300:
            self.x = 10
        if self.y >= 200:
            self.y = 5
        self.layout.move(self.label, self.x, self.y)  # 移动标签
    def main(self):                                   # 定义 main 方法
        gtk.main()
window = MyWindow('PyGTK', 300, 200)                  # 创建窗口对象
window.main()
```

运行 PyGTKLayout.py 脚本后，与上例相同，将显示如图 14-11 所示的初始界面，不同的是，单击【Move】按钮一次，标签将向右下移动，不停地单击【Move】按钮，标签将不停地向右下方移动。当标签移动超出窗口右下方边框时，将出现在窗口右上方，如图 14-13 所示。

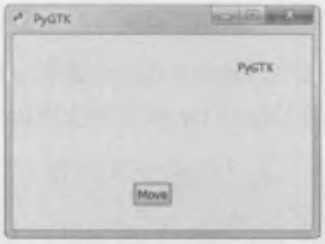


图 14-13 移动标签位置至窗口右上方

14.2.4 文本框

PyGTK 中单行文本框的创建相对较为简单，但创建多行文本框则略为烦琐。多行文本框中的内容需要使用 gtk.TextBuffer 对象进行设置或者操作，而且多行文本框一般需要与滚动窗口组合使用。

1. 单行文本框

使用 gtk.Entry()可以创建单行文本框。单行文本框具有以下几种常用的方法。

- set_visibility() 将单行文本框设置为密码框。
- get_visibility() 判断单行文本框是否为密码框。
- set_max_length() 设置单行文本框所能输入的最长字符数。
- get_max_length() 获取单行文本框所能输入的最长字符数。
- set_text() 设置单行文本框中的文字。
- get_text() 获取单行文本框中的文字。

下面所示的 PyGTKEntry.py 脚本是使用 gtk.Entry()创建了一个单行文本框和一个密码框。

```
# -*- coding:utf-8 -*-
# file: PyGTKEntry.py
#
import pygtk                                                    # 导入 pygtk 模块
pygtk.require('2.0')                                            # 设置 pygtk 所需的 gtk 版本
import gtk                                                      # 导入 gtk 模块
class MyWindow():                                               # 定义窗口类
    def __init__(self, title, width, height):                   # 定义初始化方法
        self.window = gtk.Window()                              # 生成窗口对象
        self.window.set_title(title)                            # 设置窗口标题
        self.window.set_default_size(width, height)             # 设置窗口大小
        self.window.connect('destroy', lambda q: gtk.main_quit()) # 关闭窗口时退出程序
        vbox = gtk.VBox(False, 5)                               # 生成竖向 Box 对象
        label1 = gtk.Label('Nomal')                             # 创建标签
        vbox.pack_start(label1)                                 # 向 Box 对象中添加标签
        entry1 = gtk.Entry()                                    # 创建文本框
        vbox.pack_start(entry1)                                 # 向 Box 对象中添加文本框
        entry1.show()                                           # 显示文本框
        label2 = gtk.Label('Password')                          # 创建标签
```

```
        vbox.pack_start(label2)
        entry2 = gtk.Entry()                          # 创建文本框
        entry2.set_visibility(False)                  # 将文本框设置为密码框
        vbox.pack_start(entry2)
        entry2.show()
        self.window.add(vbox)                         # 向窗口添加 Box 对象
        label1.show()                                 # 显示标签
        label2.show()
        vbox.show()
        self.window.show()                            # 显示窗口
    def main(self):                                   # 定义 main 方法
        gtk.main()                                    # 调用 gtk.main 方法
window = MyWindow('PyGTK', 200, 120)                  # 创建窗口对象
window.main()
```

运行 PyGTKEntry.py 脚本后，将显示一个窗口，在其中的单行文本框中和密码框输入文字，如图 14-14 所示。

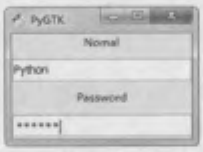


图 14-14 单行文本框和密码框

2. 多行文本框

使用 PyGTK 中的 gtk.TextView 可以创建多行文本框，其操作过程比较复杂。由于 PyGTK 中的多行文本框不含滚动条，因此常将多行文本框添加到一个滚动窗口中，以使用滚动窗口的滚动条。

当文本框创建后，应使用其 get_buffer 方法创建 gtk.TextBuffer 对象，使用 gtk.TextBuffer 对象可以获得文本框中的内容，或者对文本框中的内容进行操作。

gtk.TextView 常用的方法如下。

- set_buffer() 设置 gtk.TextBuffer 对象。
- get_buffer() 获得 gtk.TextBuffer 对象。
- set_wrap_mode() 设置换行方式。
- get_wrap_mode() 获得换行方式。
- set_editable() 设置文本框中文本的可编辑性。
- get_editable() 获得文本框中的文本是否可编辑。
- set_justification() 设置对齐方式。
- get_justification() 获得对齐方式。
- set_left_margin() 设置左边距。
- get_left_margin() 获得左边距。
- set_right_margin() 设置右边距。
- get_right_margin() 获得右边距。
- set_tabs() 设置制表符大小。
- get_tabs() 获取制表符大小。

gtk.TextBuffer 的常用方法如下。

- get_line_count()　　获取 TextBuffer 中的行数。
- get_char_count()　　获取 TextBuffer 中的字符数。
- set_text()　　设置 TextBuffer 中的文本。
- insert()　　插入文本。
- insert_at_cursor()　　在光标处插入文本。
- get_text()　　获得 TextBuffer 中的文本。

下面所示的 PyGTKTextView.py 脚本创建了一个多行文本框。

```
# -*- coding:utf-8 -*-
# file: PyGTKTextView.py
#
import pygtk                                                    # 导入 pygtk 模块
pygtk.require('2.0')                                            # 设置 pygtk 所需的 gtk 版本
import gtk                                                      # 导入 gtk 模块
class MyWindow():                                               # 定义窗口类
    def __init__(self, title, width, height):                   # 定义初始化方法
        self.window = gtk.Window()                              # 生成窗口对象
        self.window.set_title(title)                            # 设置窗口标题
        self.window.set_default_size(width, height)             # 设置窗口大小
        self.window.connect('destroy', lambda q: gtk.main_quit()) # 关闭窗口时退出程序
        vbox = gtk.VBox(False, 5)                               # 生成竖向 Box 对象
        swindow = gtk.ScrolledWindow()
        text = gtk.TextView()                                   # 创建多行文本框
        textbuffer = text.get_buffer()                          # 文本框缓冲区
        swindow.add(text)
           swindow.show()
        vbox.pack_start(swindow)                                # 向 Box 对象中添加文本框
        text.show()                                             # 显示文本框
        self.window.add(vbox)                                   # 向窗口添加 Box 对象
        vbox.show()
        self.window.show()                                      # 显示窗口
    def main(self):                                             # 定义 main 方法
        gtk.main()                                              # 调用 gtk.main 方法
window = MyWindow('PyGTK', 300, 200)                            # 创建窗口对象
window.main()
```

运行 PyGTKTextView.py 脚本后，将创建如图 14-15 所示的多行文本框。此时在多行文本框中已经实现了右键快捷菜单，在窗口中单击鼠标右键，将弹出如图 14-16 所示的右键快捷菜单。



图 14-15　多行文本框

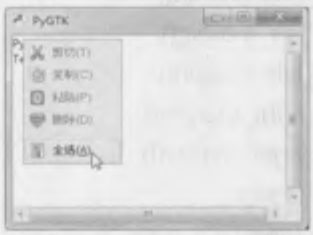


图 14-16　文本框中的右键快捷菜单

14.2.5 单选框和复选框

使用 gtk.RadioButton 可以创建单选框，其初始化参数如下。

- group　　单选框所在的组。
- label　　单选框显示的文本。
- use_underline　可选参数，若为真，则表示设置单选框文本中位于下画线之后的字母为快捷键。

当使用 gtk.RadioButton 创建单选框对象后，可以使用单选框对象的 get_active 方法判断单选框是否被选中。使用 gtk.CheckButton 可以创建复选框，它不具有 group 初始化参数，其余初始化参数与 gtk.RadioButton 的初始化参数相同。当使用 gtk.CheckButton 创建复选框对象后，也可以使用复选框对象的 get_active 方法判断复选框是否被选中。

下面所示的 PyGTKRCbutton.py 脚本创建了一组单选框和一个复选框。

```
# -*- coding:utf-8 -*-
# file: PyGTKRCbutton.py
#
import pygtk                                              # 导入 pygtk 模块
pygtk.require('2.0')                                      # 设置 pygtk 所需的 gtk 版本
import gtk                                                # 导入 gtk 模块
class MyWindow():                                         # 定义窗口类
    def __init__(self, title, width, height):             # 定义初始化方法
        self.window = gtk.Window()                        # 生成窗口对象
        self.window.set_title(title)                      # 设置窗口标题
        self.window.set_default_size(width, height)       # 设置窗口大小
        self.window.connect('destroy', lambda q: gtk.main_quit())
        self.fixed = gtk.Fixed()
        self.label1 = gtk.Label('PyGTK')                  # 创建标签
        self.fixed.put(self.label1, 80, 20)               # 添加标签
        self.label2 = gtk.Label('PyGTK')
        self.fixed.put(self.label2, 160, 20)              # 添加单选框
        self.radio1 = gtk.RadioButton(None, 'Radio1')
        self.fixed.put(self.radio1, 50, 60)
        self.radio2 = gtk.RadioButton(self.radio1, 'Radio2')
        self.fixed.put(self.radio2, 50, 90)
        self.radio3 = gtk.RadioButton(self.radio1, 'Radio3')
        self.fixed.put(self.radio3, 50, 120)
        self.check = gtk.CheckButton('CheckButton')
        self.fixed.put(self.check, 150, 60)
        self.button = gtk.Button('Test')                  # 创建按钮
        self.button.connect('clicked',self.OnButton, 'Test')   # 绑定按钮事件
        self.fixed.put(self.button, 120, 150)             # 添加按钮
        self.window.add(self.fixed)                       # 向窗口中添加 Fixed
        self.label1.show()                                # 显示组件
        self.label2.show()
        self.radio1.show()
        self.radio2.show()
        self.radio3.show()
```

```
        self.check.show()
        self.button.show()
        self.fixed.show()
        self.window.show()
    def OnButton(self, widget, data):
        if self.check.get_active():                    # 判断复选框是否选中
            self.label2.set_text('checked')
        else:
            self.label2.set_text('uncheck')
        if self.radio1.get_active():                   # 判断复选框选中状态
            self.label1.set_text('Radio1')
        elif self.radio2.get_active():
            self.label1.set_text('Radio2')
        else:
            self.label1.set_text('Radio3')
    def main(self):                                    # 定义 main 方法
        gtk.main()
window = MyWindow('PyGTK', 300, 200)                  # 创建窗口对象
window.main()
```

运行 PyGTKRCbutton.py 脚本后，将显示如图 14-17 所示窗口，在窗口中单击选择单选框【Radio2】，并选中【CheckButton】复选框，然后单击【Test】按钮，窗口上方将根据单选框和复选框的状态分别设置按钮标签文字，如图 14-18 所示。



图 14-17　创建的单选框和复选框



图 14-18　按钮标签文字发生改变

14.3 消息框和对话框

与前几章介绍的 GUI 编程工具包不同的是，PyGTK 提供的消息框和对话框的风格是 Linux 下的风格，与 Windows 系统的风格有所不同。下面将介绍 PyGTK 中的消息框和对话框。

14.3.1 消息框

使用 gtk.MessageDialog 可以创建消息框，当创建消息框后，需要调用消息框对象的 run 方法才能显示消息框，最后还应该调用消息框的 destroy 方法销毁消息框。

gtk.MessageDialog 的原型如下。

```
gtk.MessageDialog(parent, flags, type, buttons_NONE, message)
```

其参数含义如下。

- parent　　消息框的父窗口，可以为 None。
- flags　　消息框的创建标志，可以为 0。

- type　　消息框类型。
- buttons　　消息框中的按钮。
- message　　消息框中所显示的文本。

下面所示的 PyGTKMessage.py 脚本是使用 gtk.MessageDialog 创建消息框。

```
# -*- coding:utf-8 -*-
# file: PyGTKMessage.py
#
import pygtk                                                    # 导入 pygtk 模块
pygtk.require('2.0')                                            # 设置 pygtk 所需的 gtk 版本
import gtk                                                      # 导入 gtk 模块
class MyWindow():                                               # 定义窗口类
    def __init__(self, title, width, height):                   # 定义初始化方法
        self.window = gtk.Window()                              # 生成窗口对象
        self.window.set_title(title)                            # 设置窗口标题
        self.window.set_default_size(width, height)             # 设置窗口大小
        self.window.connect('destroy', lambda q: gtk.main_quit())
        self.fixed = gtk.Fixed()
        self.label = gtk.Label('MessageDialog Example')# 创建标签
        self.fixed.put(self.label, 80, 20)                      # 添加标签
        self.button = gtk.Button('Create')                      # 创建按钮
        self.button.connect('clicked',self.OnButton, 'Create') # 绑定按钮事件
        self.fixed.put(self.button, 120, 150)                   # 添加按钮
        self.window.add(self.fixed)                             # 向窗口中添加 Fixed
        self.label.show()                                       # 显示组件
        self.button.show()
        self.fixed.show()
        self.window.show()
    def OnButton(self, widget, data):                           # 按钮事件处理函数
        msg = gtk.MessageDialog(self.window,                    # 创建消息框
                gtk.DIALOG_MODAL,                               # 消息框标志
                gtk.MESSAGE_INFO,                               # 消息框类型
                gtk.BUTTONS_OK,                                 # 消息框按钮
                'An example')                                   # 消息框中的内容
        msg.run()                                               # 显示消息框
            msg.destroy()                                       # 销毁消息框
    def main(self):                                             # 定义 main 方法
        gtk.main()
window = MyWindow('PyGTK', 300, 200)                           # 创建窗口对象
window.main()
```

运行 PyGTKMessage.py 脚本后，将显示如图 14-19 所示窗口，单击【Create】按钮，将显示如图 14-20 所示的消息框。



图 14-19 创建的窗口



图 14-20 显示消息框

14.3.2 标准对话框

PyGTK 中提供了标准的文件打开、保存对话框，颜色选择对话框和字体选择对话框等。使用 gtk.FileChooserDialog 可以创建文件打开、保存对话框。使用 gtk.ColorSelectionDialog 可以创建颜色选中对话框。使用 gtk.FontSelectionDialog 可以创建字体选择对话框。

使用 gtk.FileChooserDialog 时，可以使用 gtk.FileFilter 过滤文件，即设置只查看某一类型的文件。gtk.FileFilter 有以下几种常用方法。

- set_name() 设置文件类型名。
- get_name() 获得文件类型名。
- add_pattern() 文件类型的后缀，如果是 Python 文件则写成 "*.py" 的形式。

当创建好 gtk.FileFilter 后，即可使用文件打开、关闭对话框的 add_filter 对象将其添加到对话框中。

下面所示的 PyGTKStandardDialog.py 脚本分别创建了上述标准对话框。

```
# -*- coding:utf-8 -*-
# file: PyGTKStandardDialog.py
#
import pygtk                                              # 导入 pygtk 模块
pygtk.require('2.0')                                      # 设置 pygtk 所需的 gtk 版本
import gtk                                                # 导入 gtk 模块
class MyWindow():                                         # 定义窗口类
    def __init__(self, title, width, height):             # 定义初始化方法
        self.window = gtk.Window()                        # 生成窗口对象
        self.window.set_title(title)                      # 设置窗口标题
        self.window.set_default_size(width, height)       # 设置窗口大小
        self.window.connect('destroy', lambda q: gtk.main_quit())
        self.fixed = gtk.Fixed()
        self.label1 = gtk.Label('StandardDialog Example')  # 创建标签
        self.fixed.put(self.label1, 80, 40)               # 添加标签
        self.button1 = gtk.Button('FileChooser')          # 创建按钮
        self.button1.connect('clicked',self.OnButton1, 'FileChooser') # 绑定按钮事件
        self.button2 = gtk.Button('FontChooser')          # 创建按钮
        self.button2.connect('clicked',self.OnButton2, 'FontChooser') # 绑定按钮事件
        self.button3 = gtk.Button('ColorChooser')         # 创建按钮
        self.button3.connect('clicked',self.OnButton3, 'ColorChooser') # 绑定按钮事件
        self.fixed.put(self.button1, 10, 130)             # 添加按钮
```

```
        self.fixed.put(self.button2, 95, 130)                    # 添加按钮
        self.fixed.put(self.button3, 190, 130)                   # 添加按钮
        self.window.add(self.fixed)                              # 向窗口中添加 Fixed
        self.label1.show()                                       # 显示组件
        self.button1.show()
        self.button2.show()
        self.button3.show()
        self.fixed.show()
        self.window.show()
    def OnButton1(self, widget, data):                           # 按钮事件处理函数
        dialog = gtk.FileChooserDialog('Open',                   # 创建文件打开对话框
                None,                                            # 设置父窗口
                gtk.FILE_CHOOSER_ACTION_OPEN,                    # 设置对话框标志
                (gtk.STOCK_CANCEL,                               # 添加 Cancel 按钮
                gtk.RESPONSE_CANCEL,                             # Cancel 按钮的返回值
                gtk.STOCK_OPEN,                                  # 添加 Open 按钮
                gtk.RESPONSE_OK))                                # Open 按钮的返回值
        filter = gtk.FileFilter()                                # 生成 gtk.FileFilter 对象
        filter.set_name('All files')                             # 添加文件类型名
        filter.add_pattern('*')                                  # 即所有文件
        dialog.add_filter(filter)                                # 向对话框中添加
gtk.FileFilter 对象
        filter = gtk.FileFilter()                                # 生成 gtk.FileFilter 对象
        filter.set_name('Python')                                # 添加文件类型名
        filter.add_pattern('*.py')                               # 添加文件后缀名
        filter.add_pattern('*.pyw')                              # 添加文件后缀名
        dialog.add_filter(filter)                                # 向窗口中添加
gtk.FileFilter 对象
        r = dialog.run()                                         # 显示对话框
        if r == gtk.RESPONSE_OK:
            print dialog.get_filename()
        dialog.destroy()                                         # 销毁对话框
    def OnButton2(self, widget, data):                           # 按钮事件处理函数
        fontdlg = gtk.FontSelectionDialog('Choose Font')   # 创建字体选中对话框
        r = fontdlg.run()                                        # 显示对话框
        if r == gtk.RESPONSE_OK:
            print fontdlg.get_font_name()
        fontdlg.destroy()                                        # 销毁对话框
    def OnButton3(self, widget, data):                           # 按钮事件处理函数
            colordlg = gtk.ColorSelectionDialog('Choose Color')# 创建颜色选择对话框
        colordlg.colorsel.set_has_palette(True)                  # 显示调色板
            response = colordlg.run()                            # 显示对话框
            if response == gtk.RESPONSE_OK:
               print colordlg.colorsel.get_current_color()
            colordlg.destroy()                                   # 销毁对话框
    def main(self):                                              # 定义 main 方法
        gtk.main()
window = MyWindow('PyGTK', 300, 200)                            # 创建窗口对象
window.main()
```

运行 PyGTKStandardDialog.py 脚本后，将显示如图 14-21 所示窗口。单击【FileChooser】按钮，将显示如图 14-22 所示的文件打开对话框。单击【FontChooser】按钮将显示如图 14-23 所示的字体选择对话框。单击【ColorChooser】按钮，将显示如图 14-24 所示的颜色选择对话框。



图 14-21　窗口



图 14-22　文件打开对话框



图 14-23　字体选择对话框



图 14-24　颜色选择对话框

14.3.3　自定义对话框

在 PyGTK 中使用 gtk.Dialog 可以创建自定义对话框，所创建的对话框对象和窗口对象一样可以向其中添加组件。由 gtk.Dialog 所创建的对话框被分为两部分：一部分为 VBox 对象区域，称之为“VBox”，可以向其中添加其他的组件；另一部分为 HBox 对象区域，称之为“action_area”，对话框中的按钮在该区域。gtk.Dialog 的原型如下。

```
gtk.Dialog(title=None, parent=None, flags=0, buttons=None)
```

其参数含义如下。

- ◆ title　　对话框的标题。
- ◆ parent　　对话框的父窗口。

- flags　　对话框的创建标志。
- buttons　　对话框中的按钮。

其中 buttons 参数为一个元组，由按钮 ID 及按钮返回值成对组成。

下面所示的 PyGTKDialog.py 脚本是使用 gtk.Dialog 创建自定义对话框。

```
# -*- coding:utf-8 -*-
# file: PyGTKDialog.py
#
import pygtk                                          # 导入 pygtk 模块
pygtk.require('2.0')                                  # 设置 pygtk 所需的 gtk 版本
import gtk                                            # 导入 gtk 模块
class MyWindow():                                     # 定义窗口类
    def __init__(self, title, width, height):         # 定义初始化方法
        self.window = gtk.Window()                    # 生成窗口对象
        self.window.set_title(title)                  # 设置窗口标题
        self.window.set_default_size(width, height)   # 设置窗口大小
        self.window.connect('destroy', lambda q: gtk.main_quit())
        self.fixed = gtk.Fixed()
        self.label = gtk.Label('Dialog Example')      # 创建标签
        self.fixed.put(self.label, 80, 40)            # 添加标签
        self.button = gtk.Button('CreateDialog')      # 创建按钮
        self.button.connect('clicked',self.OnButton, 'CreateDialog') # 绑定按钮事件
        self.fixed.put(self.button, 80, 130)          # 添加按钮
        self.window.add(self.fixed)                   # 向窗口中添加 Fixed
        self.label.show()                             # 显示组件
        self.button.show()
        self.fixed.show()
        self.window.show()
    def OnButton(self, widget, data):                 # 按钮事件处理函数
        dialog = gtk.Dialog('PyGTK',                  # 创建对话框
                None,                                 # 对话框父窗口
                gtk.DIALOG_MODAL,                     # 对话框标志
                        (gtk.STOCK_CANCEL,            # 向对话框中添加 Cancel 按钮
                gtk.RESPONSE_CANCEL,                  # Cancel 按钮的返回值
                        gtk.STOCK_OK,                 # 向对话框中添加 Ok 按钮
                gtk.RESPONSE_OK))                     # Ok 按钮的返回值
        fixed = gtk.Fixed()                           # 创建 Fixed 组件
        dialog.vbox.pack_start(fixed)          # 向对话框中的 vbox 添加 Fixed 组件
        label = gtk.Label('Input')                    # 创建标签
        fixed.put(label,10,5)                         # 向 Fixed 组件中添加标签
        entry = gtk.Entry()                           # 创建文本框
        fixed.put(entry,50,5)                         # 向 Fixed 组件中添加文本框
        fixed.show()                                  # 显示 Fixed 组件
        label.show()
        entry.show()
        r = dialog.run()                              # 显示对话框并获取其返回值
        if r == gtk.RESPONSE_OK:                # 如果单击 Ok 按钮则输出文本框中的内容
```

```
            print entry.get_text()
        dialog.destroy()
    def main(self):                                    # 定义 main 方法
        gtk.main()
window = MyWindow('PyGTK', 300, 200)                  # 创建窗口对象
window.main()
```

运行 PyGTKDialog.py 脚本后，将显示如图 14-25 所示窗口，单击【CreateDialog】按钮，将显示如图 14-26 所示的自定义对话框。



图 14-25　创建的窗口

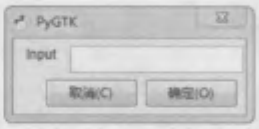


图 14-26　自定义对话框

14.4　使用菜单

PyGTK 中提供了两种创建菜单的方式，一种方式是逐个创建菜单，另一种方式是先定义好所有菜单项，然后再创建菜单。在 PyGTK 中，菜单的组织方式类似于目录的组件方式，如果预先定义菜单，则需要在菜单中使用 "/" 以表明菜单的位置。本节介绍在 PyGTK 中创建菜单的具体操作。

14.4.1　创建菜单

使用 PyGTK 中的 gtk.MenuBar、gtk.Menu、gtk.MenuItem 可以依次创建菜单。如果一次要创建较多的菜单，则可以使用 gtk.ItemFactory。

1. 依次创建菜单

使用 gtk.MenuBar 可以创建一个菜单条，当创建好菜单条后，可以使用 append 方法向其中添加下拉菜单。使用 gtk.Menu 可以创建下拉菜单，当创建好下拉菜单后，还可以使用 append 方法向其中添加菜单命令。使用 gtk.MenuItem 可以创建菜单命令。

下面所示的 PyGTKMenuItem.py 脚本创建了简单的菜单。

```
# -*- coding:utf-8 -*-
# file: PyGTKMenuItem.py
#
import pygtk                                           # 导入 pygtk 模块
pygtk.require('2.0')                                   # 设置 pygtk 所需的 gtk 版本
import gtk                                             # 导入 gtk 模块
class MyWindow():                                      # 定义窗口类
    def __init__(self, title, width, height):          # 定义初始化方法
        window = gtk.Window()                          # 生成窗口对象
        window.set_title(title)                        # 设置窗口标题
```

```
        window.set_default_size(width, height)      # 设置窗口大小
        window.connect('destroy', lambda q: gtk.main_quit())   # 关闭窗口，退出程序
        fixed = gtk.Fixed()                         # 创建 Fixed 组件
        window.add(fixed)
        filemenu = gtk.Menu()                       # 创建菜单
        open = gtk.MenuItem('Open')                 # 创建 Open 菜单命令
        open.show()                                 # 显示 Open
        close = gtk.MenuItem('Close')               # 创建 Close 菜单命令
        close.show()                                # 显示 Close
        filemenu.append(open)                       # 向菜单中添加 Open
        filemenu.append(close)                      # 向菜单中添加 Close
        file = gtk.MenuItem('_File')                # 生成 File 菜单，下画线表示快捷键
        file.set_submenu(filemenu)                  # 向 File 菜单添加项
        file.show()                                 # 显示 File 菜单
        editmenu = gtk.Menu()                       # 创建菜单
        copy = gtk.MenuItem('Copy')                 # 创建 Copy 菜单命令
        copy.show()                                 # 显示 Copy
        paste = gtk.MenuItem('Paste')               # 创建 Paste 菜单命令
        paste.show()                                # 显示 Paste
        editmenu.append(copy)                       # 向菜单中添加 Copy
        editmenu.append(paste)                      # 向菜单中添加 Paste
        edit = gtk.MenuItem('_Edit')                # 生成 Edit 菜单
        edit.set_submenu(editmenu)                  # 向 Edit 菜单中添加项
        edit.show()                                 # 显示 Edit 菜单
        menubar = gtk.MenuBar()                     # 生成菜单条
        menubar.append(file)                        # 向菜单条中添加 File 菜单
        menubar.append(edit)                        # 向菜单条中添加 Edit 菜单
        fixed.put(menubar, 0, 0)                    # 向 Fixed 组件中添加菜单条
        menubar.show()                              # 显示菜单条组件
        fixed.show()
        window.show()
    def main(self):                                 # 定义 main 方法
        gtk.main()
window = MyWindow('PyGTK', 300, 200)                # 创建窗口对象
window.main()
```

运行 PyGTKMenuItem.py 脚本后，可以看到如图 14-27 所示的【File】菜单和如图 14-28 所示的【Edit】菜单。



图 14-27 【File】菜单



图 14-28 【Edit】菜单

2. 使用 gtk.ItemFactory 创建菜单

如果要创建的菜单项比较多，则可以使用 gtk.ItemFactory 和 gtk.MenuBar 来创建。使用 gtk.ItemFactory 创建菜单时，首先要用元组的形式定义菜单。元组中的每一项为一个元组，其中包含了菜单的位置、快捷键、菜单命令的回调函数、回调函数动作值（默认为 0）和菜单的样式。图 14-27 所示的菜单可以写成如下形式的元组。

```
(
    ( '/_File',     None,   None, 0, '<Branch>' ),
    ( '/File/Open', None,   None, 0, None ),
    ( '/File/Save', None,   None, 0, None )
)
```

菜单的位置类似于文件的路径，不同的是这里用的是 Linux 下的 "/" 表示。位于下画线之后的字母表示可以使用<Alt+字母>进行快捷键访问该菜单项。如果在菜单项中使用了快捷键，则还应该使用 gtk.AccelGroup 创建快捷键对象，并将其传递给 gtk.ItemFactory，然使用 Windows 对象的 accel_group()将快捷键添加到窗口中。

使用 gtk.ItemFactory 创建菜单时需向其传递所创建的菜单类型、菜单路径和快捷键对象，菜单路径应为 "<main>"。

下面所示的 PyGTKMenu.py 脚本是使用 gtk.ItemFactory 创建了一组菜单。

```
# -*- coding:utf-8 -*-
# file: PyGTKMenu.py
#
import pygtk                                              # 导入 pygtk 模块
pygtk.require('2.0')                                      # 设置 pygtk 所需的 gtk 版本
import gtk                                                # 导入 gtk 模块
class MyWindow():                                         # 定义窗口类
    def __init__(self, title, width, height):             # 定义初始化方法
        window = gtk.Window()                             # 生成窗口对象
        window.set_title(title)                           # 设置窗口标题
        window.set_default_size(width, height)            # 设置窗口大小
        window.connect('destroy', lambda q: gtk.main_quit())   # 关闭窗口，退出程序
        fixed = gtk.Fixed()                               # 创建 Fixed 组件
        window.add(fixed)
        menu_items = (                                    # 菜单
                    ( '/_File',     None,         None, 0, '<Branch>' ),
                    ( '/File/Open',  '<control>O', None, 0, None ),
                    ( '/File/Save',  '<control>S', None, 0, None ),
                    ( '/File/s',    None,         None, 0, '<Separator>' ),
                    ( '/File/Close', '<control>Q', None, 0, None ),
                    ( '/_Edit',      None,         None, 0, '<Branch>' ),
                    ( '/Edit/Copy',  None,         None, 0, None ),
                    ( '/Edit/Paste',  None,           None, 0, None ),
                    ( '/Edit/s',    None,         None, 0, '<Separator>' ),
                    ( '/Edit/Cut',      None,         None, 0, None ),
                    ( '/_Help',     None,         None, 0, '<Branch>' ),
                    ( '/Help/About',    None,         None, 0, None ),
                    )
        accel_group = gtk.AccelGroup()                    # 创建快捷键对象
```

```
        itemfactory = gtk.ItemFactory(gtk.MenuBar,          # 创建 ItemFactory 对象
                    '<main>', accel_group)
        itemfactory.create_items(menu_items)                # 从菜单元组创建菜单
        window.add_accel_group(accel_group)                 # 向窗口中添加快捷键
        menubar = gtk.MenuBar()                             # 生成菜单条
        menubar = itemfactory.get_widget('<main>')          # 获得菜单
        fixed.put(menubar, 0, 0)                            # 向 Fixed 组件中添加菜单条
        menubar.show()                                      # 显示菜单条组件
        fixed.show()
        window.show()
    def main(self):                                         # 定义 main 方法
        gtk.main()
window = MyWindow('PyGTK', 300, 200)                       # 创建窗口对象
window.main()
```

运行 PyGTKMenu.py 脚本后，单击【File】菜单，显示如图 14-29 所示；单击【Edit】菜单，显示如图 14-30 所示，单击【Help】菜单，显示如图 14-31 所示。

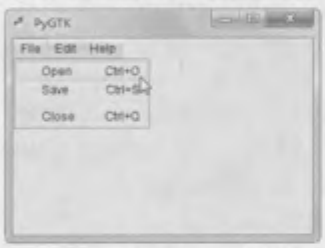


图 14-29 【File】菜单

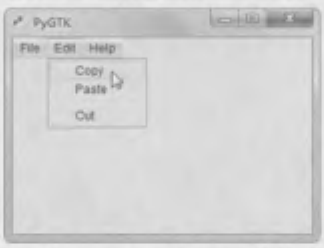


图 14-30 【Edit】菜单

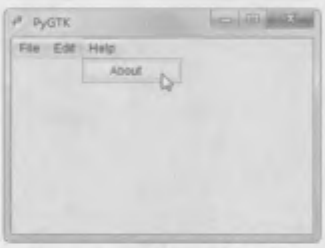


图 14-31 【Help】菜单

运行脚本后将出现如下所示的警告。

```
E:\book\code\\PyGTKMenu.py:30: DeprecationWarning: use gtk.UIManager
  item_factory = gtk.ItemFactory(gtk.MenuBar, '<main>', accel_group)
```

这是因为菜单项直接写在脚本中而没有使用 XML 格式的文件，出现该警告并不影响程序运行，可以忽略。

14.4.2 菜单事件

菜单事件的绑定与按钮事件的绑定类似，只需使用菜单对象的 connect 方法绑定菜单事件的响应函数即可，菜单事件的响应函数应该按照 connect 方法所使用的参数来定义，不同的是，菜单事件的事件名为“activate”。

下面所示的 PyMenuEvent.py 脚本是使用菜单的 connect 方法绑定菜单事件。

```
# -*- coding:utf-8 -*-
# file: PyMenuEvent.py
#
import pygtk                                                # 导入 pygtk 模块
pygtk.require('2.0')                                        # 设置 pygtk 所需的 gtk 版本
import gtk                                                  # 导入 gtk 模块
class MyWindow():                                           # 定义窗口类
    def __init__(self, title, width, height):               # 定义初始化方法
```

```
        window = gtk.Window()                                    # 生成窗口对象
        window.set_title(title)                                  # 设置窗口标题
        window.set_default_size(width, height)                   # 设置窗口大小
        window.connect('destroy', lambda q: gtk.main_quit())     # 关闭窗口，退出程序
        fixed = gtk.Fixed()                                      # 创建 Fixed 组件
        window.add(fixed)
        filemenu = gtk.Menu()                                    # 创建菜单
        open = gtk.MenuItem('Open')                              # 创建 Open 菜单命令
        open.show()                                              # 显示 Open
        open.connect('activate', self.OnOpen, 'Open')            # 绑定菜单事件
        close = gtk.MenuItem('Close')                            # 创建 Close 菜单命令
        close.connect('activate', self.OnClose, 'Close')         # 绑定菜单事件
        close.show()                                             # 显示 Close
        filemenu.append(open)                                    # 向菜单中添加 Open
        filemenu.append(close)                                   # 向菜单中添加 Close
        file = gtk.MenuItem('_File')                             # 生成 File 菜单，下画线表示快捷键
        file.set_submenu(filemenu)                               # 向 File 菜单添加项
        file.show()                                              # 显示 File 菜单
        menubar = gtk.MenuBar()                                  # 生成菜单条
        menubar.append(file)                                     # 向菜单条中添加 File 菜单
        fixed.put(menubar, 0, 0)                                 # 向 Fixed 组件中添加菜单条
        menubar.show()                                           # 显示菜单条组件
        fixed.show()
        window.show()
    def OnOpen(self, widget, data):                              # 处理菜单事件
        dialog = gtk.FileChooserDialog('Open',                   # 创建打开文件对话框
                    None,
                    gtk.FILE_CHOOSER_ACTION_OPEN,
                    (gtk.STOCK_CANCEL, gtk.RESPONSE_CANCEL,
                    gtk.STOCK_OPEN, gtk.RESPONSE_OK))
        dialog.set_default_response(gtk.RESPONSE_OK)
        response = dialog.run()
        dialog.destroy()
    def OnClose(self, widget, data):                             # 处理菜单事件
        gtk.main_quit()                                          # 退出程序
    def main(self):                                              # 定义 main 方法
        gtk.main()
window = MyWindow('PyGTK', 300, 200)                            # 创建窗口对象
window.main()
```

14.5 资源文件

在 PyGTK 中，可以使用两种形式的资源文件：一种是文本形式的，类似于 Windows 下资源文件的格式；另一种则是 XML 形式，类似于 wxPython 中的资源文件格式。由于 XML 格式的资源文件可以使用 Glade 创建，使用十分简便，因此，本节以 XML 形式的资源为例进行讲解。

14.5.1 使用 Glade 创建资源文件

Glade 是可视化界面设计工具，它用于生成 GTK 的界面代码。Glade 生成的资源文件同样可以被 PyGTK 所使用。使用 Glade 可以简化程序，而且 Glade 功能十分强大，在脚本中使用 gtk.glade 模块可以十分方便地创建 GUI 界面。

1. 安装 Glade

Glade 是开源软件，可以从其官方 http://glade.gnome.org/网站下载源代码进行编译安装。Windows 用户可以到 http://ftp.gnome.org/pub/GNOME/binaries/win32/glade/下载编译好的 Windows 版本的 Glade。

由于已经安装了 PyGTK，因此只需下载 Glade，而不必下载 GTK+的运行库。如果没有安装 PyGTK，但是需要使用 Glade，则必须下载 GTK+程序的运行库，才可以使用 Glade。

http://ftp.gnome.org/pub/GNOME/binaries/win32/glade/提供了 3.8 和 3.14 两个版本供下载。以 glade-3-14-2-installer.exe 为例，下载完成后，双击该程序即可进行安装。首先将显示如图 14-32 所示的界面，单击【I Agree】按钮，显示如图 14-33 所示界面，设置安装位置，然后单击【Install】按钮开始安装。



图 14-32 许可协议



图 14-33 设置安装位置

安装完成后，单击 Windows 7 的【开始】菜单，依次选择【GNOME Foundation】|【Glade Interface Designer】即可启动 Glade，显示如图 14-34 所示界面。

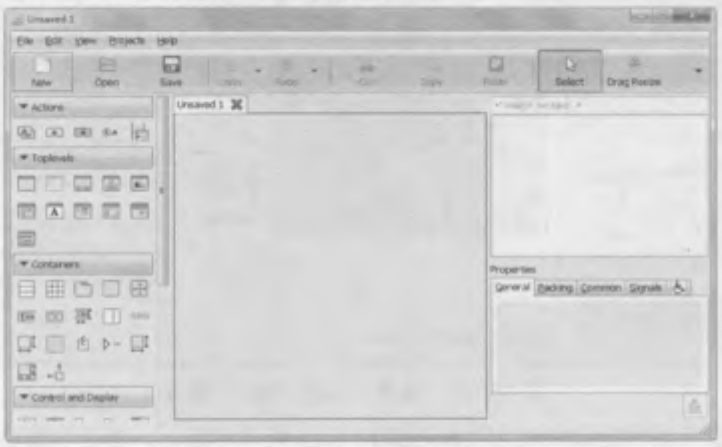


图 14-34 Glade 窗口

从图 14-34 中可以看出，与一般的 Windows 应用程序不同，Glade 由三个窗口组成：一个工程管理窗口（位于中间的窗口）、一个常用组件窗口（位于左侧的窗口）和一个属性管理窗口（位于右侧的窗口）。

2. 创建资源文件

Glade 的使用与 VS2008 中的资源编辑器类似，创建一个资源文件的步骤如下。

step 1 单击【File】|【New】命令，新建一个工程。

step 2 单击左侧常用组件窗口中的 Window 按钮，新建一个窗口，如图 14-35 所示。

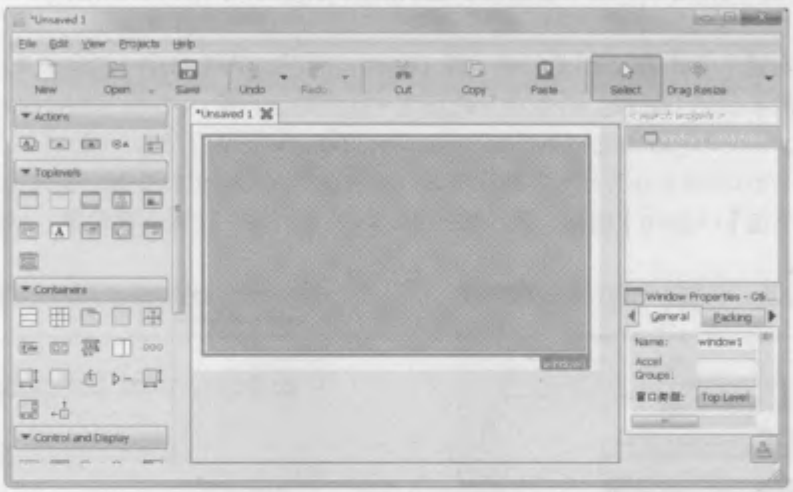


图 14-35 新建的 window 窗口

step 3 在右侧属性窗口中的【Name】文本框中输入窗口名“window”，在脚本中即可使用“window”载入所创建的资源。在【窗口标题】中输入窗口的标题“PyGTK Glade”，如图 14-36 所示。



图 14-36 设置 window 窗口的属性

step 4 在左侧常用组件窗口中找到并单击选择 Flex 控件按钮，然后在新建的 window 窗口中单击，向窗口中添加 Fixed 组件。

step 5 单击左侧常用组件窗口中的 Menu Bar 按钮 File，然后在窗口中单击，向窗口中添加一个标准菜单。重新调整菜单的位置和大小，如图 14-37 所示。

step 6 单击常用组件窗口中的 TextView 按钮，然后单击图 14-37 所示的窗口，向窗口中添加 TextView 组件。重新调整其大小，如图 14-38 所示。



图 14-37 向窗口中添加菜单



图 14-38 向窗口中添加多行文本框

step 7 双击图 14-38 所示窗口中的【File】菜单，选中弹出菜单中的【退出】命令。此时属性窗口如图 14-39 所示。单击属性窗口中的【Signals】标签，选中【GtkMenuItem】|【activate】项。在【Handler】一列中填入“OnQuit”，在【User data】一列中填入“Quit”，绑定【Quit】菜单事件，如图 14-40 所示。

step 8 单击菜单【File】|【Save】命令，将工程保存为“res.glade”。



图 14-39 【退出】菜单命令的属性窗口



图 14-40 绑定【Quit】菜单事件

14.5.2 使用资源文件

在脚本中，使用 Glade 创建的资源文件时只需载入资源文件、载入主窗口即可。如果在资源文件中添加了事件信号，则在脚本中也应该载入。在脚本中使用资源文件时，主要是使用 gtk.glade.XML 类的以下几个方法。

- get_widget() 获得资源文件中的组件。
- signal_connect() 绑定信号事件。
- signal_autoconnect() 绑定多个信号事件。

对于 signal_autoconnect()方法，其参数为一个字典。以 14.5.1 节所添加的【Quit】命令的信

号为例，将其写在参数字典中，关键字为"OnQuit"，值为事件处理函数。下面所示的 PyGTKGlade.py 脚本是使用 gtk.glade 模块来使用资源文件。

```
# -*- coding:utf-8 -*-
# file: PyGTKGlade.py
#
import pygtk
pygtk.require('2.0')
import gtk
import gtk.glade
class MyWindow():                                          # 定义窗口类
    def __init__(self):                                    # 定义初始化方法
        res = gtk.glade.XML('res.glade')                   # 载入资源文件，生成
gtk.glade.XML 实例
        window = res.get_widget('window')                  # 载入 window
        signal = { 'OnQuit' : self.OnQuit }                # 创建信号字典
        res.signal_autoconnect(signal)                     # 绑定信号事件
        window.connect('destroy', lambda q:gtk.main_quit())# 窗口关闭则退出程序
        window.show()
    def OnQuit(self, widget):                              # Quit 菜单命令处理事件
        gtk.main_quit()
    def main(self):                                        # 定义 main 方法
        gtk.main()                                         # 调用 gtk.main 方法
window = MyWindow()                                        # 创建窗口对象
window.main()
```

运行 PyGTKGlade.py 脚本后，将创建如图 14-41 所示的窗口。



图 14-41　使用 Glade 资源文件创建窗口

14.6 本章小结

本章介绍了使用 PyGTK 模块编写 GUI 的方法。由于 PyGTK 模块目前还不支持 Python 3，因此本章以 Python 2.7 为基础介绍了 PyGTK 的使用。与前几章介绍的 GUI 设计模块类似，本章也介绍了使用 PyGTK 模块的组件、消息框和标准对话框的方法，同时还介绍了自定义对话框、菜单及菜单事件的使用。与前几章不同的是，PyGTK 中提供了一种可视化资源文件设计器，可用可视化的方式的创建 GUI，然后在 Python 脚本中调用。

在下一章中将介绍一种 GUI 设计模块 PyQT 的使用。

◆　◆　◆

第 15 章 使用 PyQT 编写 GUI

本章包括

- PyQt 概述
- 使用 PyQt 消息框和标准对话框
- 创建和使用资源文件
- 使用组件
- 创建自定义对话框

Qt 库是一个面向对象的图形用户界面库，可以在多个操作系统上使用。PyQt 是一个创建 GUI 应用程序的工具包，它是 Python 编程语言和 Qt 库的成功融合。PyQt 实现了一个 Python 模块集，它拥有超过 300 类，将近 6000 个函数和方法。PyQt 是一个多平台的工具包，可以运行在所有主要操作系统上，包括 UNIX，Windows 和 Mac。

15.1 PyQt 概述

Qt 是比较早的图形用户界面库，但 Qt 具有较为严格的许可证限制。PyQt 具有和 Qt 一样的许可证，PyQt 采用双许可证，即 GPL 和商业许可。在此之前，GPL 的版本只能用在 UNIX 上，从 PyQt 的版本 4 开始，GPL 许可证可用于所有支持的平台。

15.1.1 PyQt 的安装

PyQt 的官方网站是 http://www.riverbankcomputing.co.uk/，首先需要从该网站下载适用的安装程序包。目前网站下载首页提供的是支持 Python 2.7 和 Python 3.3 的安装包。不过，在下载首页提供了一个其他版的下载链接，单击进入 sourceforge.net 网站，找到适用于 Python 3.2 版的最新版本为 PyQt-4.10，具体链接地址为 http://sourceforge.net/projects/pyqt/files/PyQt4/PyQt-4.10/，找到适用于 Windows 下 Python 3.2 的安装包 PyQt4-4.10-gpl-Py3.2-Qt4.8.4-x32.exe，单击下载。

由于是一个完整的安装包，因此安装操作较为简单，安装过程如下。

step 1 用鼠标双击运行安装文件 PyQt4-4.10-gpl-Py3.2-Qt4.8.4-x32.exe，显示如图 15-1 所示的安装向导界面。

step 2 单击【Next】按钮进入下一步安装，显示如图 15-2 所示的许可协议界面。

step 3 单击【I Agree】按钮，同意许可协议，显示如图 15-3 所示的选择安装组件界面。

step 4 在图 15-3 所示界面中，默认是选中了所有的组件，直接单击【Next】按钮，显示如图 15-4 所示的选择 Python 3.2 安装位置的界面，安装程序默认已查找到 Python 3.2 的安装位置。如果这里的路径不正确，则需要进行修改。



图 15-1 Qt 安装程序



图 15-2 许可协议



图 15-3 选择安装组件

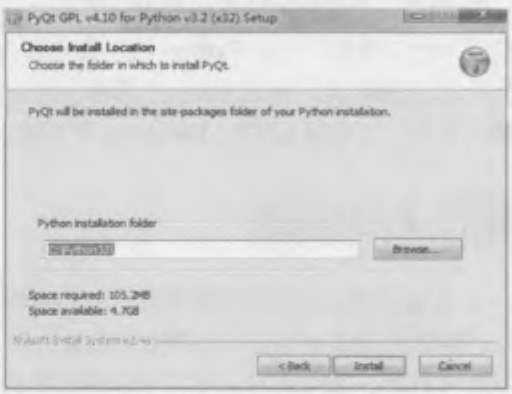


图 15-4 安装 MingW

step 5 单击【Install】按钮开始安装各组件，经过一段时间后，安装完成，单击对话框中的【Finish】按钮，完成 PyQt 的安装。

安装完成后，在 Windows 7 的【开始】菜单中将增加一个名为【PyQt GPL v4.10 for Python v3.2 (x32)】的程序组，其中提供了 PyQt 一些工具，如资源编辑器、演示程序、文档等。

15.1.2 使用 PyQt 创建窗口

下面编写一个简短的程序，测试安装是否正确。

使用 PyQt 创建 GUI 脚本时，应首先创建一个 QtGui.QApplication 对象，并向其传递命令行参数。因此，在脚本中除了导入 PyQt 模块外，还应该导入 sys 模块，并且在脚本的最后应调用 QtGui.QApplication 对象的 exec_方法进入消息循环。

下面所示的 HelloPyQt.py 脚本创建了一个简单的窗口，以测试 PyQt 是否安装正确。

```
# -*- coding:utf-8 -*-
# file: HelloPyQt.py
#
import sys                                          # 导入 sys 模块
from PyQt4 import QtCore, QtGui                     # 导入 PyQt 模块
class MyWindow(QtGui.QMainWindow):                  # 通过继承 QtGui.QMainWindow 创建类
    def __init__(self):                             # 初始化方法
```

```
        QtGui.QMainWindow.__init__(self)          # 调用父类初始化方法
        self.setWindowTitle('PyQt')               # 设置窗口标题
        self.resize(300,200)                      # 设置窗口大小
app = QtGui.QApplication(sys.argv)                # 创建 QApplication 对象
mywindow = MyWindow()                             # 创建 MyWindow 对象
mywindow.show()                                   # 显示窗口
app.exec_()                                       # 进入消息循环
```

运行 HelloPyQt.py 脚本后，将创建如图 15-5 所示的窗口。



图 15-5 使用 PyQt 创建的窗口

15.2 组件

PyQt 提供了丰富的组件，使用这些组件可以方便地进行 GUI 编程。在 PyQt 中可以方便地使用组件，并使用信号/插槽进行组件之间的通信，处理组件事件。

15.2.1 标签

在 PyQt 中，使用 QtGui.QLabel 可以创建标签，使用其 setText 方法可以设置标签中的文字，使用 setTextFormat 方法可以设置标签中文字的格式，当创建标签后，可以使用 QMainWindow 的 setCentralWidget 方法将标签添加到窗口中。QtGui.QLabel 的常用方法如下。

- setPicture() 设置标签中的图片。
- setText() 设置标签中的文字。
- setTextFormat() 设置标签中文本的格式。
- setAlignment() 设置标签中文本的对齐方式。

下面所示的 PyQtLabel.py 脚本是使用 QtGui.QLabel 创建了一个标签。

```
# -*- coding:utf-8 -*-
# file: PyQtLabel.py
#
import sys                                        # 导入 sys 模块
from PyQt4 import QtCore, QtGui                   # 导入 PyQt 模块
class MyWindow(QtGui.QMainWindow):                # 通过继承
QtGui.QMainWindow 创建类
    def __init__(self):                           # 初始化方法
        QtGui.QMainWindow.__init__(self)          # 调用父类初始化方法
```

```
        self.setWindowTitle('PyQt')                          # 设置窗口标题
        self.resize(300,200)                                 # 设置窗口大小
        label = QtGui.QLabel('PyQt\nLabel')                  # 创建标签
        label.setAlignment(QtCore.Qt.AlignCenter)            # 设置标签文字对齐样式
        self.setCentralWidget(label)                         # 向窗口中添加标签
app = QtGui.QApplication(sys.argv)                           # 创建 QApplication 对象
mywindow = MyWindow()                                        # 创建 MyWindow 对象
mywindow.show()                                              # 显示窗口
app.exec_()                                                  # 进入消息循环
```

运行 PyQtLabel.py 脚本后，将创建如图 15-6 所示的含一个标签窗口。



图 15-6　含一个标签的窗口

15.2.2　布局组件和空白项

在窗口中使用 setCentralWidget 只能添加一个组件，如果需要向窗口中添加多个组件，则需要使用布局组件来组织这些组件。使用布局组件不仅可以容纳多个组件，还可以设置组件的位置。

1. 布局组件

布局组件主要用于控件其内部组件的大小、位置等。布局组件可以包含其他的组件，也可以包含其他的布局组件。常用的布局组件如下。

- QLayout　　基本的布局组件，只能被继承。
- QHBoxLayout　　横向 Box 布局组件。
- QVBoxLayout　　竖向 Box 布局组件。
- QGridLayout　　Grid 布局组件。

布局组件共有的方法有以下几个。

- addWidget()　　添加组件。
- addLayout()　　添加其他布局组件。

下面所示的 PyQtLayout.py 脚本是使用布局组件来布置多个标签。

```
# -*- coding:utf-8 -*-
# file: PyQtLayout.py
#
import sys                                                   # 导入 sys 模块
from PyQt4 import QtCore, QtGui                              # 导入 PyQt 模块
class MyWindow(QtGui.QWidget):                               # 通过继承 QtGui.QWidget 创建类
```

```
    def __init__(self):                                    # 初始化方法
        QtGui.QWidget.__init__(self)                       # 调用父类初始化方法
        self.setWindowTitle('PyQt')                        # 设置窗口标题
        self.resize(300,200)                               # 设置窗口大小
        gridlayout = QtGui.QGridLayout()                   # 创建布局组件
        hboxlayout1 = QtGui.QHBoxLayout()
        hboxlayout2 = QtGui.QHBoxLayout()
        vboxlayout1 = QtGui.QVBoxLayout()
        vboxlayout2 = QtGui.QVBoxLayout()
        label1 = QtGui.QLabel('Label1',self)               # 创建标签
        label1.setAlignment(QtCore.Qt.AlignCenter)
        label2 = QtGui.QLabel('Label2')
        label3 = QtGui.QLabel('Label3')
        label4 = QtGui.QLabel('Label4')
        label5 = QtGui.QLabel('Label5')
        hboxlayout1.addWidget(label1)                      # 向布局组件中添加标签
        vboxlayout1.addWidget(label2)
        vboxlayout1.addWidget(label3)
        vboxlayout2.addWidget(label4)
        vboxlayout2.addWidget(label5)
        hboxlayout2.addLayout(vboxlayout1)          # 向 hboxlayout2 添加 vboxlayout1
        hboxlayout2.addLayout(vboxlayout2)          # 向 hboxlayout2 添加 vboxlayout2
        gridlayout.addLayout(hboxlayout1, 0 ,0)     # 向第一行第一列添加 hboxlayout1
        gridlayout.addLayout(hboxlayout2, 1, 0)     # 向第二行第一列添加 hboxlayout2
        gridlayout.setRowMinimumHeight (1, 180)     # 设置第二行的最小高度为 108
        self.setLayout(gridlayout)
app = QtGui.QApplication(sys.argv)                         # 创建 QApplication 对象
mywindow = MyWindow()                                      # 创建 MyWindow 对象
mywindow.show()                                            # 显示窗口
app.exec_()                                                # 进入消息循环
```

运行 PyQtLayout.py 脚本后，将创建如图 15-7 所示的含多个标签的窗口。



图 15-7 创建的含多个标签窗口

2. 空白项

PyQt 中的空白项可以占据位置，这样就可以更好地控制其他组件的位置。

使用 QtGui.QSpacerItem 可以创建空白项，创建空白项时，可以使用宽度和高度设置空白项的大小。当创建空白项后，需使用布局组件的 addItem 方法将空白项添加到布局组件中。

下面所示的 PyQtSpacer.py 脚本是使用空白项布置组件。

```
# -*- coding:utf-8 -*-
# file: PyQtSpacer.py
#
import sys                                          # 导入 sys 模块
from PyQt4 import QtCore, QtGui                     # 导入 PyQt 模块
class MyWindow(QtGui.QWidget):                      # 通过继承 QtGui.QWidget 创建类
    def __init__(self):                             # 初始化方法
        QtGui.QWidget.__init__(self)                # 调用父类初始化方法
        self.setWindowTitle('PyQt')                 # 设置窗口标题
        self.resize(300,200)                        # 设置窗口大小
        gridlayout = QtGui.QGridLayout()            # 创建布局组件
        spacer1 = QtGui.QSpacerItem(300,40)         # 创建空白项
        spacer2 = QtGui.QSpacerItem(300,40)
        label = QtGui.QLabel('Label',self)          # 创建标签
        label.setAlignment(QtCore.Qt.AlignCenter)
        gridlayout.addItem(spacer1, 0 ,0)           # 添加空白项
        gridlayout.addWidget(label, 1, 0)           # 添加标签
        gridlayout.addItem(spacer2, 2, 0)           # 添加空白项
        self.setLayout(gridlayout)
app = QtGui.QApplication(sys.argv)                  # 创建 QApplication 对象
mywindow = MyWindow()                               # 创建 MyWindow 对象
mywindow.show()                                     # 显示窗口
app.exec_()                                         # 进入消息循环
```

运行 PyQtSpacer.py 脚本后，将创建如图 15-8 所示的使用空白项布置组件的窗口。



图 15-8　创建的使用空白项布置组件的窗口

15.2.3 按钮

使用 PyQt 中的 QtGui.QPushButton 可以创建按钮，在 PyQt 中，按钮事件是通过信号/插槽的形式将按钮事件绑定到类的方法上的。

1. 创建按钮

当使用 QtGui.QPushButton 创建按钮后，可以使用以下几种方法设置按钮的样式、属性等。

- setDefault()　　将按钮设置为默认按钮。
- setFlat()　　将按钮设置为平坦模式。
- setMenu()　　设置按钮所关联的菜单。
- menu()　　获得按钮所关联的菜单。

下面所示的 PyQtButton.py 脚本是使用 QtGui.QPushButton 创建了两个按钮。

```
# -*- coding:utf-8 -*-
# file: PyQtButton.py
#
import sys                                          # 导入 sys 模块
from PyQt4 import QtCore, QtGui                     # 导入 PyQt 模块
class MyWindow(QtGui.QWidget):                      # 通过继承 QtGui.QWidget 创建类
    def __init__(self):                             # 初始化方法
        QtGui.QWidget.__init__(self)                # 调用父类初始化方法
        self.setWindowTitle('PyQt')                 # 设置窗口标题
        self.resize(300,200)                        # 设置窗口大小
        gridlayout = QtGui.QGridLayout()            # 创建布局组件
        button1 = QtGui.QPushButton('Button1')      # 生成 Button1
        gridlayout.addWidget(button1, 1, 1, 1 ,3)   # 添加 Button1
        button2 = QtGui.QPushButton('Button2')      # 生成 Button2
        button2.setFlat(True)
        gridlayout.addWidget(button2, 2, 2)         # 添加 Button2
        self.setLayout(gridlayout)                  # 向窗口中添加布局组件
app = QtGui.QApplication(sys.argv)                  # 创建 QApplication 对象
mywindow = MyWindow()                               # 创建 MyWindow 对象
mywindow.show()                                     # 显示窗口
app.exec_()                                         # 进入消息循环
```

运行 PyQtButton.py 脚本后，将创建如图 15-9 所示的窗口。



图 15-9 创建按钮

2. 信号和插槽

Qt 中的组件是使用信号和插槽的形式来进行通信。Qt 的组件中有很多预定义的信号，当事件激发时，组件会发出信号，信号被发送给插槽进行处理。插槽实际上是处理特定信号的函数，在 PyQt 中也是如此，可以使用组件的 connect 方法将组件信号绑定到其处理插槽上。connect 方法的原型如下。

```
connect (QObject, SIGNAL(), SLOT(), Qt.ConnectionType)
```

其参数含义如下。

- QObject　　发送信号的组件。
- SIGNAL()　　组件所发送的信号。
- SLOT()　　插槽函数。
- Qt.ConnectionType　　可选参数，连接类型。

下面所示的 PyQtButtonEvent.py 脚本是使用 connect 方法将按钮的“clicked()”信号连接到事件处理插槽函数上。

```
# -*- coding:utf-8 -*-
# file: PyQtButtonEvent.py
#
import sys
from PyQt4 import QtCore, QtGui
class MyWindow(QtGui.QWidget):
    def __init__(self):
        QtGui.QWidget.__init__(self)
        self.setWindowTitle('PyQt')                          # 设置窗口标题
        self.resize(300,200)                                 # 设置窗口大小
        gridlayout = QtGui.QGridLayout()                     # 创建布局组件
        self.button1 = QtGui.QPushButton('Button1')          # 生成 Button1
        gridlayout.addWidget(self.button1, 1, 1, 1 ,3)
        self.button2 = QtGui.QPushButton('Button2')          # 生成 Button2
        gridlayout.addWidget(self.button2, 2, 2)
        self.setLayout(gridlayout)                           # 向窗口中添加布局组件
        self.connect(self.button1,                           # Button1 事件
                QtCore.SIGNAL('clicked()'),                  # clicked()信号
                self.OnButton1)                              # 插槽函数
        self.connect(self.button2,                           # Button2 事件
                QtCore.SIGNAL('clicked()'),                  # clicked()信号
                self.OnButton2)                              # 插槽函数
    def OnButton1(self):                                     # Button1 插槽函数
        self.button1.setText('clicked')
    def OnButton2(self):                                     # Button2 插槽函数
        self.button2.setText('clicked')
app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()
```

运行 PyQtButtonEvent.py 脚本后，将显示如图 15-10 左图所示窗口，单击窗口中的两个按钮后，可以看到按钮上显示的文本已经发生了变化，如右图所示。



图 15-10　按钮事件

15.2.4 文本框

PyQt 中提供了单行文本框和多行文本框，使用 QtGui.QLineEdit 可以创建单行文本框，使用 QtGui.QTextEdit 可以创建多行文本框。

1. 单行文本框

使用 QtGui.QLineEdit 可以创建单行文本框，通过修改其属性可以将其设置为密码框。当创建单行文本框后，可以使用以下方法设置文本框的属性或对文本框中的内容进行操作。

- clear() 清除文本框中的内容。
- contextMenuEvent() 右键菜单事件。
- copy() 复制文本框中的内容。
- cut() 剪切文本框中的内容。
- paste() 向文本框中粘贴内容。
- redo() 重做。
- selectAll() 全选。
- selectedText() 获得选中的文本。
- setAlignment() 设置文本对齐方式。
- setEchoMode() 设置文本框类型。
- setMaxLength() 设置文本框中最大字符数。
- setText() 设置文本框中的文字。
- undo() 撤销。

下面所示的 PyQtLineEdit.py 脚本创建了一个单行文本框和一个密码框。

```
# -*- coding:utf-8 -*-
# file: PyQtLineEdit.py
#
import sys
from PyQt4 import QtCore, QtGui
class MyWindow(QtGui.QWidget):                          # 通过继承 QtGui.QWidget 创建类
    def __init__(self):                                 # 初始化方法
        QtGui.QWidget.__init__(self)                    # 调用父类初始化方法
        self.setWindowTitle('PyQt')                     # 设置窗口标题
        self.resize(300,200)                            # 设置窗口大小
        gridlayout = QtGui.QGridLayout()                # 创建布局组件
        label1 = QtGui.QLabel('Normal Lineedit')        # 创建标签
        label1.setAlignment(QtCore.Qt.AlignCenter)
        gridlayout.addWidget(label1, 0, 0 )
        edit1 = QtGui.QLineEdit()                       # 创建单行文本框
        gridlayout.addWidget(edit1, 1, 0)
        label2 = QtGui.QLabel('Password')               # 创建标签
        label2.setAlignment(QtCore.Qt.AlignCenter)
        gridlayout.addWidget(label2, 2, 0)
        edit2 = QtGui.QLineEdit()                       # 创建单行文本框
        edit2.setEchoMode(QtGui.QLineEdit.Password)# 将其设置为密码框
        gridlayout.addWidget(edit2, 3, 0)
        self.setLayout(gridlayout)
app = QtGui.QApplication(sys.argv)                      # 创建 QApplication 对象
mywindow = MyWindow()                                   # 创建 MyWindow 对象
mywindow.show()                                         # 显示窗口
```

```
app.exec_()                                    # 进入消息循环
```

运行 PyQtLineEdit.py 脚本后，在单行文本框和密码框中输入字符，将得到如图 15-11 所示的效果。

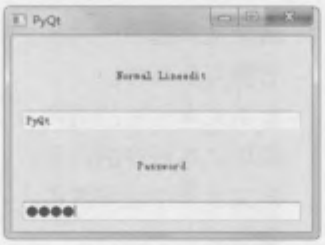


图 15-11　创建单行文本框

2. 多行文本框

使用 QtGui.QTextEdit 可以创建多行文本框，当创建多行文本框后，可以使用以下方法设置文本框的属性或对文本框中的内容进行操作。

- append()　　向文本框中追加内容。
- clear()　　清除文本框中的内容。
- contextMenuEvent()　　右键菜单事件。
- copy()　　复制文本框中的内容。
- cut()　　剪切文本框中的内容。
- find()　　查找文本。
- paste()　　向文本框中粘贴内容。
- redo()　　重做。
- selectAll()　　全选。
- selectedText()　　获得选中的文本。
- setAlignment()　　设置文本对齐方式。
- setText()　　设置文本框中的文字。
- undo()　　撤销。

下面所示的 PyQtTextEdit.py 脚本创建了一个多行文本框。

```
# -*- coding:utf-8 -*-
# file: PyQtTextEdit.py
#
import sys
from PyQt4 import QtCore, QtGui
class MyWindow(QtGui.QWidget):
    def __init__(self):                                    # 初始化方法
        QtGui.QWidget.__init__(self)                       # 调用父类初始化方法
        self.setWindowTitle('PyQt')                        # 设置窗口标题
        self.resize(300,200)                               # 设置窗口大小
        gridlayout = QtGui.QGridLayout()                   # 创建布局组件
        label = QtGui.QLabel('TextEdit')                   # 创建标签
        label.setAlignment(QtCore.Qt.AlignCenter)
```

```
        gridlayout.addWidget(label, 0, 0 )
        edit = QtGui.QTextEdit()                          # 创建多行文本框
        edit.setText('Python\nPyQt')                      # 设置文本框中的文字
        gridlayout.addWidget(edit, 1, 0)
        self.setLayout(gridlayout)
app = QtGui.QApplication(sys.argv)                        # 创建 QApplication 对象
mywindow = MyWindow()                                     # 创建 MyWindow 对象
mywindow.show()                                           # 显示窗口
app.exec_()                                               # 进入消息循环
```

运行 PyQtTextEdit.py 脚本后，将创建如图 15-12 所示的窗口，在多行文本框中可输入多行文本。



图 15-12 创建的多行文本框

15.2.5 单选框和复选框

在 PyQt 中，使用 QtGui.QRadioButton 可以创建单选框，使用 QtGui.QCheckBox 可以创建复选框。单选框和复选框都可以通过 setChecked()设置状态，通过 isChecked()方法获取状态。

下面所示的 PyQtRCButton.py 脚本创建了一组单选框和一个复选框。

```
# -*- coding:utf-8 -*-
# file: PyQtRCButton.py
#
import sys
from PyQt4 import QtCore, QtGui
class MyWindow(QtGui.QWidget):
    def __init__(self):                                   # 初始化方法
        QtGui.QWidget.__init__(self)                      # 调用父类初始化方法
        self.setWindowTitle('PyQt')                       # 设置窗口标题
        self.resize(300,200)                              # 设置窗口大小
        gridlayout = QtGui.QGridLayout()                  # 创建布局组件
        self.label1 = QtGui.QLabel('Label1')              # 创建标签
        self.label2 = QtGui.QLabel('Label2')
        gridlayout.addWidget(self.label1, 1, 2)
        gridlayout.addWidget(self.label2, 2, 2)
        self.radio1 = QtGui.QRadioButton('Radio1')        # 创建单选框
        self.radio2 = QtGui.QRadioButton('Radio2')
        self.radio3 = QtGui.QRadioButton('Radio3')
        self.radio1.setChecked(True)                      # 将 Radio1 选中
        gridlayout.addWidget(self.radio1, 1, 1 )          # 添加单选框
        gridlayout.addWidget(self.radio2, 2, 1 )
        gridlayout.addWidget(self.radio3, 3, 1 )
```

```
        self.check = QtGui.QCheckBox('check')                    # 创建复选框
        self.check.setChecked(True)                              # 将复选框选中
        gridlayout.addWidget(self.check, 3, 2)
        self.button = QtGui.QPushButton('Test')                  # 创建按钮
        gridlayout.addWidget(self.button, 4, 1, 1, 2)
        self.setLayout(gridlayout)                               # 向窗口中添加布局组件
        self.connect(self.button,                                # 按钮事件
                QtCore.SIGNAL('clicked()'),
                self.OnButton)
    def OnButton(self):                                          # 按钮插槽函数
        if self.radio1.isChecked():                              # 判断单选框是否选中
            self.label1.setText('Radio1')
        elif self.radio2.isChecked():
            self.label1.setText('Radio2')
        else :
            self.label1.setText('Radio3')
        if self.check.isChecked():                               # 判断复选框是否选中
            self.label2.setText('checked')
        else:
            self.label2.setText('uncheck')
app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()
```

运行 PyQtRCButton.py 脚本后，在窗口中选择单选按钮和复选框，然后单击【Test】按钮，将根据单选框和复选框的状态设置窗口右侧标签中的文本，如图 15-13 所示。



图 15-13　单选框和复选框

15.2.6　菜单

在 PyQt 中，可以使用 QMenuBar 创建菜单条，使用 QMenu 创建菜单。菜单单击事件也可以像按钮事件一样，通过信号/插槽的形式绑定到类的方法上。

1. 创建菜单

当使用 QMenuBar 创建菜单条后，可以使用 addMenu 方法向菜单条中添加菜单。然后通过菜单的 addAction 方法向菜单中添加菜单命令。

对于 QtGui.QMainWindow，可以直接使用其 menuBar 方法获得其菜单条，而无须使用 QMenuBar 创建菜单条。

下面所示的 PyQtMenu.py 脚本创建了一组菜单。

```
# -*- coding:utf-8 -*-
# file: PyQtMenu.py
```

```
#
import sys
from PyQt4 import QtCore, QtGui
class MyWindow(QtGui.QMainWindow):                    # 通过继承
QtGui.QMainWindow 创建类
    def __init__(self):                               # 初始化方法
        QtGui.QMainWindow.__init__(self)              # 调用父类初始化方法
        self.setWindowTitle('PyQt')                   # 设置窗口标题
        self.resize(300,200)                          # 设置窗口大小
        menubar = self.menuBar()                      # 获得窗口的菜单条
        file = menubar.addMenu('&File')               # 添加 File 菜单
        file.addAction('Open')                        # 添加 Open 命令
        file.addAction('Save')                        # 添加 Save 命令
        file.addSeparator()                           # 添加分隔符
        file.addAction('Close')                       # 添加 Close 命令
        edit = menubar.addMenu('&Edit')               # 添加 Edit 菜单
        edit.addAction('Copy')                        # 添加 Copy 命令
        edit.addAction('Paste')                       # 添加 Paste 命令
        edit.addAction('Cut')                         # 添加 Cut 命令
        edit.addAction('SelectAll')                   # 添加 SelectAll 命令
        help = menubar.addMenu('&Help')               # 添加 Help 菜单
        help.addAction('About')                       # 添加 About 命令
app = QtGui.QApplication(sys.argv)                    # 创建 QApplication 对象
mywindow = MyWindow()                                 # 创建 MyWindow 对象
mywindow.show()                                       # 显示窗口
app.exec_()                                           # 进入消息循环
```

运行 PyQtMenu.py 脚本后，将创建如图 15-14、图 15-15 和图 15-16 所示的菜单。



图 15-14 File 菜单

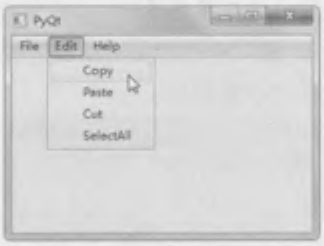


图 15-15 Edit 菜单



图 15-16 Help 菜单

2. 菜单事件

菜单事件的处理和按钮事件的处理一样，都是使用窗口的 connect 方法将菜单信号绑定到插槽事件处理函数上。不同的是，对于菜单事件，通常需要绑定其“triggered()”信号。

下面所示的 PyQtMenuAction.py 脚本是对菜单事件进行处理，并且创建了右键菜单。

```
# -*- coding:utf-8 -*-
# self.file: PyQtMenuAction.py
#
import sys
from PyQt4 import QtCore, QtGui
class MyWindow(QtGui.QMainWindow):
```

```
    def __init__(self):                                    # 初始化方法
        QtGui.QMainWindow.__init__(self)                   # 调用父类初始化方法
        self.setWindowTitle('PyQt')                        # 设置窗口标题
        self.resize(300,200)                               # 设置窗口大小
        self.label = QtGui.QLabel('Menu Action')           # 创建标签
        self.label.setAlignment(QtCore.Qt.AlignCenter) # 设置标签文字对齐样式
        self.setCentralWidget(self.label)
        menubar = self.menuBar()                           # 获得窗口的菜单条
        self.file = menubar.addMenu('&File')               # 添加 File 菜单
        open = self.file.addAction('Open')                 # 添加 Open 命令
        self.connect(open, QtCore.SIGNAL('triggered()'), self.OnOpen)  # 菜单信号
        save = self.file.addAction('Save')                 # 添加 Save 命令
        self.connect(save, QtCore.SIGNAL('triggered()'), self.OnSave)  # 菜单信号
        self.file.addSeparator()                           # 添加分隔符
        close = self.file.addAction('Close')               # 添加 Close 命令
        self.connect(close, QtCore.SIGNAL('triggered()'), self.OnClose)# 菜单信号
    def OnOpen(self):
        self.label.setText('Menu Action: Open')
    def OnSave(self):
        self.label.setText('Menu Action: Save')
    def OnClose(self):
        self.close()
    def contextMenuEvent(self, event):                     # 重载弹出式菜单事件
        self.file.exec_(event.globalPos())
app = QtGui.QApplication(sys.argv)                         # 创建 QApplication 对象
mywindow = MyWindow()                                      # 创建 MyWindow 对象
mywindow.show()                                            # 显示窗口
app.exec_()                                                # 进入消息循环
```

运行 PyQtMenuAction.py 脚本后，单击菜单【File】命令将显示下拉菜单，如图 15-17 所示。单击下拉菜单中的【Open】命令，在窗口中将显示一个提示信息，如图 15-18 所示。右击窗口中的空白区域，将显示如图 15-19 所示的右键菜单。



图 15-17 【File】下拉菜单



图 15-18 单击【Open】命令



图 15-19 右键菜单

15.3 创建对话框

PyQt 中提供了基本的消息框和标准对话框，可以在脚本中直接使用。另外，在 PyQt 中也可以根据需要创建自定义的对话框，并在对话框中使用 PyQt 中的组件。

15.3.1 消息框和标准对话框

使用 PyQt 提供的类和方法可以方便地创建和使用消息框、标准对话框等。标准对话框包括基本的打开、关闭文件对话框，字体选择对话框和颜色选择对话框等。

1. 消息框

使用 QtGui.QMessageBox 类中的方法可以创建简单的消息框，用于向用户传递信息。QtGui.QMessageBox 类中包含了以下几种创建消息框的方法。

- about()　　创建关于消息框。
- aboutQt()　　创建关于 Qt 消息框。
- critical()　　创建错误处理对话框。
- information()　　创建信息消息框。
- question()　　创建询问消息框。
- warning()　　创建警告消息框。

下面所示的 PyQtMessageBox.py 脚本是使用 QtGui.QMessageBox 创建消息框。

```
# -*- coding:utf-8 -*-
# file: PyQtMessageBox.py
#
import sys
from PyQt4 import QtCore, QtGui
class MyWindow(QtGui.QWidget):
    def __init__(self):                                          # 初始化方法
        QtGui.QWidget.__init__(self)                             # 调用父类初始化方法
        self.setWindowTitle('PyQt')                              # 设置窗口标题
        self.resize(300,200)                                     # 设置窗口大小
        gridlayout = QtGui.QGridLayout()                         # 创建布局组件
        self.label = QtGui.QLabel('MessBox example')
        gridlayout.addWidget(self.label, 1, 3, 1, 3)
        self.button1 = QtGui.QPushButton('About')                # 生成 Button1
        gridlayout.addWidget(self.button1, 2, 1)
        self.button2 = QtGui.QPushButton('AboutQt')              # 生成 Button2
        gridlayout.addWidget(self.button2, 2, 2)
        self.button3 = QtGui.QPushButton('Critical')             # 生成 Button3
        gridlayout.addWidget(self.button3, 2, 3)
        self.button4 = QtGui.QPushButton('Info')                 # 生成 Button4
        gridlayout.addWidget(self.button4, 2, 4)
        self.button5 = QtGui.QPushButton('Qusetion')             # 生成 Button5
        gridlayout.addWidget(self.button5, 2, 5)
        self.button6 = QtGui.QPushButton('Warning')              # 生成 Button6
        gridlayout.addWidget(self.button6, 2, 6)
        spacer = QtGui.QSpacerItem(200, 80)
        gridlayout.addItem(spacer, 3, 1, 1, 5)
        self.setLayout(gridlayout)                               # 向窗口中添加布局组件
        self.connect(self.button1,                              # Button1 事件
                QtCore.SIGNAL('clicked()'),
                self.OnButton1)
```

```
        self.connect(self.button2,                                  # Button2 事件
                QtCore.SIGNAL('clicked()'),
                self.OnButton2)
        self.connect(self.button3,                                  # Button3 事件
                QtCore.SIGNAL('clicked()'),
                self.OnButton3)
        self.connect(self.button4,                                  # Button4 事件
                QtCore.SIGNAL('clicked()'),
                self.OnButton4)
        self.connect(self.button5,                                  # Button5 事件
                QtCore.SIGNAL('clicked()'),
                self.OnButton5)
        self.connect(self.button6,                                  # Button6 事件
                QtCore.SIGNAL('clicked()'),
                self.OnButton6)
    def OnButton1(self):                                            # Button1 插槽函数
        self.button1.setText('clicked')
        QtGui.QMessageBox.about(self, 'PyQt', 'About')              # 创建 About 消息框
    def OnButton2(self):                                            # Button2 插槽函数
        self.button2.setText('clicked')
        QtGui.QMessageBox.aboutQt(self, 'PyQt')                     # 创建 AboutQt 消息框
    def OnButton3(self):                                            # Button3 插槽函数
        self.button3.setText('clicked')
        r = QtGui.QMessageBox.critical(self, 'PyQt',                # 创建 Ctitical 消息框
                'Critical',
                QtGui.QMessageBox.Abort,
                QtGui.QMessageBox.Retry,
                QtGui.QMessageBox.Ignore)
        if r == QtGui.QMessageBox.Abort:
            self.label.setText('Abort')
        elif r == QtGui.QMessageBox.Retry:
            self.label.setText('Retry')
        else:
            self.label.setText('Ignore')
    def OnButton4(self):                                            # Button4 插槽函数
        self.button4.setText('clicked')
        QtGui.QMessageBox.information(self, 'PyQt', 'Information')
                                                                    # 创建 Information 消息框
    def OnButton5(self):                                            # Button5 插槽函数
        self.button5.setText('clicked')
        r = QtGui.QMessageBox.question(self, 'PyQt',                # 创建 Question 消息框
                'Question',
                QtGui.QMessageBox.Yes,
                QtGui.QMessageBox.No,
                QtGui.QMessageBox.Cancel)
    def OnButton6(self):                                            # Button6 插槽函数
        self.button6.setText('clicked')
        r = QtGui.QMessageBox.warning(self, 'PyQt',                 # 创建 Warning 消息框
                'Warning',
                QtGui.QMessageBox.Yes,
                QtGui.QMessageBox.No)
app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()
```

运行 PyQtMessageBox.py 脚本后，将显示如图 15-20 所示的窗口，其中有多个按钮。分别单击窗口中的按钮，将依次创建如图 15-21、图 15-22、图 15-23、15-24、15-25 和图 15-26 所示的窗口。



图 15-20 演示窗口



图 15-21 About 消息框



图 15-22 AboutQt 消息框



图 15-23 Cirtical 消息框

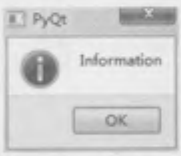


图 15-24 Information 消息框

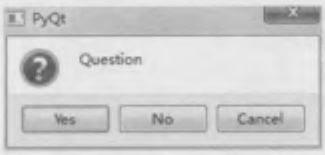


图 15-25 Question 消息框

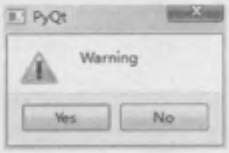


图 15-26 Warning 消息框

2. 标准对话框

在 PyQt 中，使用 QtGui.QFileDialog 提供的方法可以创建文件打开、关闭对话框；使用 QtGui.QFontDialog 提供的方法可以创建字体选择对话框；使用 QtGui.QColorDialog 提供的方法可以创建颜色选择对话框。

对于 QtGui.QfileDialog，主要有以下几个常用方法。

- getExistingDirectory() 创建选取路径对话框。
- getOpenFileName() 创建打开文件对话框。

- getOpenFileNames()　　创建打开文件对话框，可以同时打开多个文件。
- getSaveFileName()　　创建保存文件对话框。

对于 QtGui.QfontDialog，其静态方法仅有 getFont，用于创建字体选择对话框。

对于 QtGui.QColorDialog，可以使用其 getColor 方法创建颜色选择对话框。

下面所示的 PyQtStandarDialog.py 脚本创建了 PyQt 提供的标准对话框。

```
# -*- coding:utf-8 -*-
# file: PyQtStandarDialog.py
#
import sys
from PyQt4 import QtCore, QtGui
class MyWindow(QtGui.QWidget):
    def __init__(self):                                          # 初始化方法
        QtGui.QWidget.__init__(self)                             # 调用父类初始化方法
        self.setWindowTitle('PyQt')                              # 设置窗口标题
        self.resize(300,200)                                     # 设置窗口大小
        gridlayout = QtGui.QGridLayout()                         # 创建布局组件
        self.label = QtGui.QLabel('StandarDialog example')
        gridlayout.addWidget(self.label, 1, 2)
        self.button1 = QtGui.QPushButton('File')                 # 生成 Button1
        gridlayout.addWidget(self.button1, 2, 1)
        self.button2 = QtGui.QPushButton('Font')                 # 生成 Button2
        gridlayout.addWidget(self.button2, 2, 2)
        self.button3 = QtGui.QPushButton('Color')                # 生成 Button3
        gridlayout.addWidget(self.button3, 2, 3)
        spacer = QtGui.QSpacerItem(200, 80)
        gridlayout.addItem(spacer, 3, 1, 1, 3)
        self.setLayout(gridlayout)                               # 向窗口中添加布局组件
        self.connect(self.button1,                               # Button1 事件
                QtCore.SIGNAL('clicked()'),
                self.OnButton1)
        self.connect(self.button2,                               # Button2 事件
                QtCore.SIGNAL('clicked()'),
                self.OnButton2)
        self.connect(self.button3,                               # Button3 事件
                QtCore.SIGNAL('clicked()'),
                self.OnButton3)
    def OnButton1(self):
        self.button1.setText('clicked')
        filename = QtGui.QFileDialog.getOpenFileName(self, 'Open')
                                                                 # 创建文件打开对话框
        if not filename.isEmpty():
            self.label.setText(filename)
    def OnButton2(self):                                         # Button2 插槽函数
        self.button2.setText('clicked')
        font, ok = QtGui.QFontDialog.getFont()                   # 创建字体选中对话框
        if ok:
            self.label.setText(font.key())
    def OnButton3(self):                                         # Button3 插槽函数
        self.button3.setText('clicked')
        color = QtGui.QColorDialog.getColor()                    # 创建颜色选择对话框
```

```
        if color.isValid():
            self.label.setText(color.name())
app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()
```

运行 PyQtStandarDialog.py 脚本后，将显示如图 15-27 所示窗口。单击【File】按钮，将创建如图 15-28 所示的【打开文件】对话框，单击【Font】按钮，将创建如图 15-29 所示的【字体选择】对话框，单击【Color】按钮，将创建如图 15-30 所示的【颜色选择】对话框。



图 15-27 演示窗口



图 15-28 【打开文件】对话框



图 15-29 【字体选择】对话框

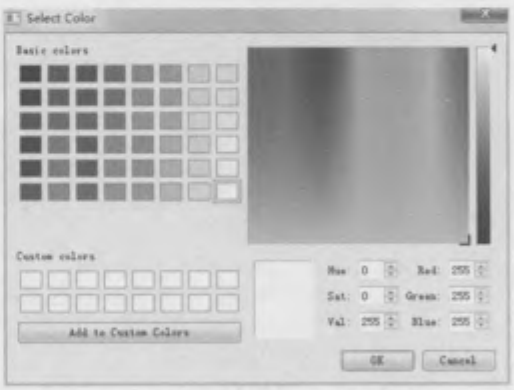


图 15-30 【颜色选择】对话框

15.3.2 自定义对话框

通过继承 QtGui.QDialog 类，可以创建自定义对话框。所创建的自定义对话框和窗口一样，可以向其中添加组件，通过使用 connect 方法可以响应组件事件。

下面所示的 PyQtDialog.py 脚本是使用 QtGui.QDialog 创建一个自定义对话框。

```
# -*- coding:utf-8 -*-
# file: PyQtDialog.py
#
```

```
import sys
from PyQt4 import QtCore, QtGui
class MyDialog(QtGui.QDialog):                                  # 继承 QtGui.QDialog
    def __init__(self):
        QtGui.QDialog.__init__(self)
        self.gridlayout = QtGui.QGridLayout()                   # 创建布局组件
        self.label = QtGui.QLabel('Input:')                     # 创建标签
        self.gridlayout.addWidget(self.label, 0, 0)
        self.edit = QtGui.QLineEdit()                           # 创建单行文本框
        self.gridlayout.addWidget(self.edit, 0, 1)
        self.ok = QtGui.QPushButton('Ok')                       # 创建 Ok 按钮
        self.gridlayout.addWidget(self.ok, 1, 0)
        self.cancel = QtGui.QPushButton('Cancel')               # 创建 Cancel 按钮
        self.gridlayout.addWidget(self.cancel, 1, 1)
        self.setLayout(self.gridlayout)
        self.connect(self.ok,                                   # Ok 按钮事件
                QtCore.SIGNAL('clicked()'),
                self.OnOk)
        self.connect(self.cancel,                               # Cancel 按钮事件
                QtCore.SIGNAL('clicked()'),
                self.OnCancel)
    def OnOk(self):                                             # 处理 Ok 按钮事件
        self.text = self.edit.text()                            # 获取文本框中内容
        self.done(1)                                            # 结束对话框返回 1
    def OnCancel(self):                                         # 处理 Cancel 按钮事件
        self.done(0)                                            # 结束对话框返回 0
class MyWindow(QtGui.QWidget):
    def __init__(self):                                         # 初始化方法
        QtGui.QWidget.__init__(self)                            # 调用父类初始化方法
        self.setWindowTitle('PyQt')                             # 设置窗口标题
        self.resize(300,200)                                    # 设置窗口大小
        gridlayout = QtGui.QGridLayout()                        # 创建布局组件
        self.button = QtGui.QPushButton('CreateDialog')         # 生成 Button1
        gridlayout.addWidget(self.button, 1, 1)
        self.setLayout(gridlayout)                              # 向窗口中添加布局组件
        self.connect(self.button,                               # Button 事件
                QtCore.SIGNAL('clicked()'),
                self.OnButton)
    def OnButton(self):                                         # 处理按钮事件
        dialog = MyDialog()                                     # 创建对话框对象
        r = dialog.exec_()                                      # 运行对话框
        if r:
            self.button.setText(dialog.text)
app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()
```

运行 PyQtDialog.py 脚本后，将显示如图 15-31 左图所示窗口，单击【CreateDialog】按钮，将创建右图所示的对话框。



图 15-31 自定义对话框

15.4 使用资源

在 Qt 中，资源文件是以“.ui”为后缀的文件。Qt 提供了 Qt Designer 用于创建资源文件（Qt Designer 是随 Qt 安装的，不必单独安装），使用 Qt Designer 创建的资源文件可以在 PyQt 中使用。使用资源文件可以简化界面设计，也可以将界面和代码分离，提高程序的可维护性。

15.4.1 使用 Qt Designer 创建资源文件

Qt Designer 是所见即所得的资源文件编辑器，使用 Qt Designer 可以方便地创建复杂的 GUI 界面。使用 PyQt 中的 uic 模块可以在 Python 脚本中载入资源文件，快捷地创建 GUI 界面。

使用 Qt Designer 创建资源文件的操作步骤如下。

step 1 单击【开始】|【所有程序】|【PyQt GPL v4.10 for Python v3.2 (x32)】|【Designer】命令，即可运行 Qt Designer，显示如图 15-32 所示的界面。



图 15-32 Qt Designer 窗口

step 2 如图 15-32 所示，启动 Designer 时，将显示一个【新建窗体】对话框，该对话框中列出了常用窗体的模板，选中第 1 项，单击对话框下方的【创建】按钮将创建一个对话框，如图 15-33 所示。



图 15-33　创建对话框

step 3　调整图 15-33 所示的对话框大小（将下方的两个按钮向上移，并拖动对话框下角，调整对话框的高度和宽度）。接着在左侧的【Widget Box】浮动窗口中的【Display Widgets】下的【Label】项上按住鼠标左键，将其拖放到所创建的对话框中，如图 15-34 所示。



图 15-34　添加标签

step 4　选择【Property Editor】浮动窗口中的【text】项，将【值】改为“Input”，即将标签显示的文本修改为“Input”，如图 15-35 所示。

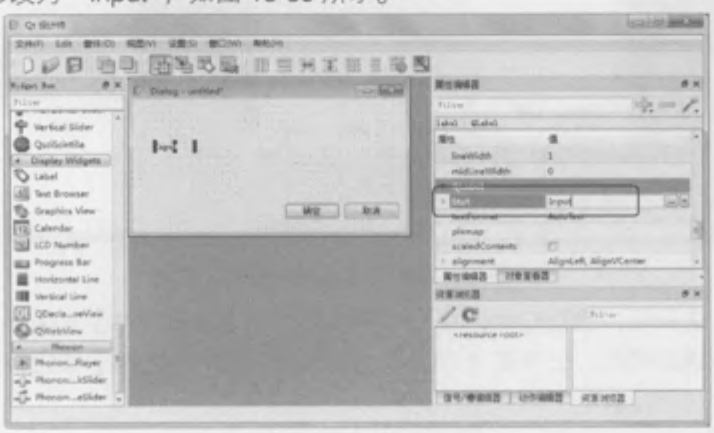


图 15-35　修改标签文本

step 5 在【Widget Box】浮动窗口中的【Input Widgets】下的【Line Edit】项上按住鼠标左键，将其拖放到所创建的对话框中，如图 15-36 所示。



图 15-36 添加单行文本框

step 6 选择【Property Editor】浮动窗口中的【Object Name】项，查看所添加的单行文本框的名字。该名字可以在脚本中使用。

step 7 单击【File】|【Save Form】命令，将资源文件保存为“res.ui”。

15.4.2 使用资源文件

使用 PyQt 中的 uic 模块可以载入资源文件中的组件，在脚本中只需使用 uic.loadUi 就可以载入资源文件。

下面所示的 PyQtRes.py 脚本是由 PyQtDialog.py 脚本改写得来的，从修改的脚本可以看出，通过使用资源文件，可简化对话框的建立。

```
# -*- coding:utf-8 -*-
# file: PyQtRes.py
#
import sys
from PyQt4 import QtCore, QtGui, uic
class MyDialog(QtGui.QDialog):                          # 继承 QtGui.QDialog
    def __init__(self):
        QtGui.QWidget.__init__(self)
        uic.loadUi("res.ui", self)                      # 载入资源文件
class MyWindow(QtGui.QWidget):
    def __init__(self):                                 # 初始化方法
        QtGui.QWidget.__init__(self)                    # 调用父类初始化方法
        self.setWindowTitle('PyQt')                     # 设置窗口标题
        self.resize(300,200)                            # 设置窗口大小
        gridlayout = QtGui.QGridLayout()                # 创建布局组件
        self.button = QtGui.QPushButton('CreateDialog') # 生成 Button1
        gridlayout.addWidget(self.button, 1, 1)
        self.setLayout(gridlayout)                      # 向窗口中添加布局组件
        self.connect(self.button,                       # Button 事件
                QtCore.SIGNAL('clicked()'),
```

```
                self.OnButton)
    def OnButton(self):                                    # 处理按钮事件
        dialog = MyDialog()                                # 创建对话框对象
        r = dialog.exec_()                                 # 运行对话框
        if r:
            self.button.setText(dialog.lineEdit.text())
app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()
```

运行 PyQtRes.py 脚本后，将显示如图 15-37 左图所示窗口，单击【CreateDialog】按钮，将创建右图所示的对话框。对比图 15-36 可以看出，这里创建的对话框就是在图 15-36 中所创建的对话框资源。



图 15-37　使用资源文件

15.5　本章小结

本章介绍了一种 GUI 设计模块 PyQT，通过这个模块可以设计出为多种操作系统使用的 GUI。PyQT 不是 Python 的标准库，因此，本章首先介绍了 PyQT 的下载和安装，然后介绍了使用 PyQT 创建窗口、使用 PyQT 组件、创建 PyQT 对话框等操作。与上一章介绍的 PyGTK 类似，在 PyQT 中也可以使用可视化设计器来设计 GUI 的资源文件，然后在 Python 脚本中调用这些资源文件。

本书前面 15 章介绍了 Python 基本语法、图形用户界面等内容，从下一章开始将进入 Python 的高级篇，下一章将学习如何使用 Python 操作数据库。

◆　　◆　　◆

Part

第 2 部分　高级篇

第 16 章　Python 与数据库

第 17 章　Python Web 应用

第 18 章　Python 网络编程

第 19 章　处理 HTML 与 XML

第 20 章　功能强大的正则表达式

第 21 章　科学计算

第 22 章　Python 扩展和嵌入

第 23 章　多线程编程



第 16 章 Python 与数据库

本章包括

- 使用 ODBC 连接 Access 数据库
- 使用 ADO 连接 Access 数据库
- 使用嵌入式数据库 SQLite
- 使用 DAO 连接 Access 数据库
- 使用 MySQL 数据库

在计算机中，经常需要保存或处理大量数据，这时就会用数据库来存储这些数据，数据库中的数据是按照一定的模型进行组织和存储的。Python 提供了对大多数数据库的支持，在 Python 中，可以进行连接到数据库、查询数据、添加删除数据等各种操作。

16.1 连接 Access 数据库

Access 数据库是 Microsoft Office 中的一部分，Access 数据库被称之为桌面数据库，适用于中小型的应用系统。Access 界面友好、操作简单，对于要求不高的应用，使用 Access 数据库是不错的选择。在 Python 中可以通过多种方式对 Access 数据库进行操作。

16.1.1 使用 ODBC 连接 Access 数据库

ODBC（Open Database Connectivity）是 Microsoft 提出的数据库访问接口标准。ODBC 提供了独立于数据库的 API 函数，使用 ODBC 可以使用统一的方式访问不同的数据库，对于不同数据库的支持是由 ODBC 的驱动层实现的。PythonWin 中的 odbc 模块提供了对 ODBC 的支持。PythonWin 中的 dbi 模块定义了各种数据类型。

1. 创建数据库

使用数据库之前首先应在 Access 中创建一个数据库，具体操作步骤如下。

step 1 打开 Access 2013，单击【文件】|【新建】命令，如图 16-1 所示，显示【新建】模板选择界面。

step 2 单击选择【空白桌面数据库】项，弹出如图 16-2 所示对话框，选择保存数据库的位置，并输入

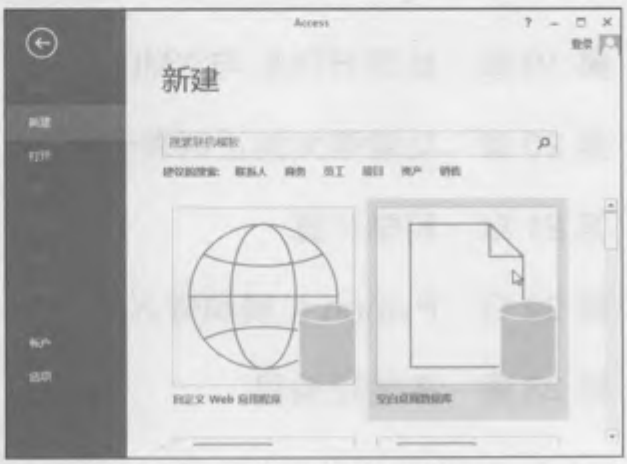


图 16-1 【新建】模板选择界面

数据库名称为“python”。



图 16-2 【空白桌面数据库】对话框

step 3 单击对话框下方的【创建】按钮后，将进入 Access 操作界面，如图 16-3 所示。

step 4 在图 16-3 所示界面中，左侧列表中显示的是数据库中已有的表，默认有一个名称为“表 1”的表，用鼠标右击“表 1”，并从弹出的快捷菜单中选择“设计视图”命令，将显示如图 16-4 所示的【另存为】对话框，将表命名为“people”。



图 16-3 Access 操作界面



图 16-4 【另存为】对话框

step 5 单击【确定】按钮，进入表设计视图，输入表的各字段，包括 ID、name、age、sex 等字段，如图 16-5 所示。



在图 16-5 所示界面中，字段“ID”前面有一个图标，表示这个字段是主键。

step 6 对各字段设置完成后，单击窗口左上角的【保存】按钮即可。

step 7 关闭表设计窗口，双击 Access 窗口左侧列表中的“people”表，将打开向表中添加数据的界面。在这个界面中添加两行数据，如图 16-6 所示。

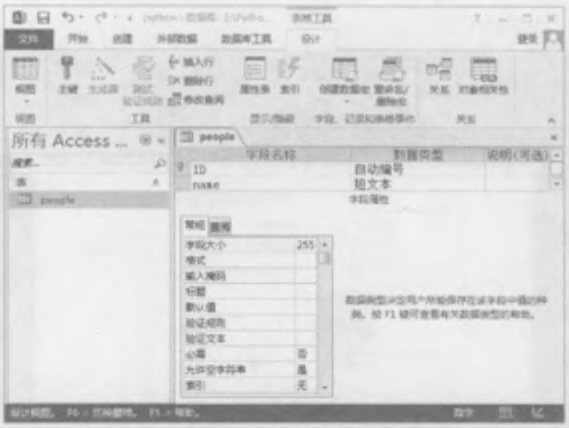


图 16-5　设计表的名字段



图 16-6　添加两行数据

step 8　单击窗口左上角的【保存】按钮，将创建的数据库保存。

2. 设置数据源

使用 ODBC 连接数据库时需要设置数据源，使其指向所创建的数据库。设置数据源的步骤如下。

step 1　单击【开始】|【控制面板】，打开 Windows 7 的【控制面板】窗口，如图 16-7 所示。

step 2　在【控制面板】窗口中找到并双击【管理工具】，打开如图 16-8 所示的【管理工具】窗口。



图 16-7　【控制面板】窗口



图 16-8　【管理工具】窗口

step 3 双击【数据源（ODBC）】命令，打开如图 16-9 所示的【ODBC 数据源管理器】对话框。

step 4 选择【用户 DSN】下【用户数据源】中的【dBASE Files】项，单击右侧的【添加】按钮，打开如图 16-10 所示的【创建新数据源】对话框。



图 16-9 【ODBC 数据源管理器】对话框



图 16-10 【创建新数据源】对话框

step 5 选择创建数据源对话框中的【Microsoft Access Driver (*.mdb, *.accdb)】选项，单击【完成】按钮，将打开如图 16-11 所示的对话框。

step 6 在【数据源名】文本框中填入“podbc”，单击【选择】按钮，将弹出【选择数据库】对话框，如图 16-12 所示，在这里选择前面创建的 Access 数据库。单击【确定】按钮，返回到图 16-11 所示的对话框。



图 16-11 设置数据库



图 16-12 【选择数据库】对话框

step 7 在图 16-11 所示对话框中单击【确定】按钮，完成 ODBC 数据源的设置。



如果使用的是 64 位的 Windows 7 系统，则需使用 C:\Windows\SysWOW64 目录中的 odbcad32.exe 来创建 ODBC 数据源。

3. 连接数据库

完成 ODBC 数据源设置以后，就可以使用 PythonWin 中的 odbc 模块访问所添加的数据库了。使用 odbc 模块时应首先使用其 odbc 方法连接到数据库，创建一个 connection 对象，然后使用 connection 对象的 cursor 方法创建一个游标，使用游标对象的 execute 方法可以执行 SQL 语句对数据库进行操作。odbc 模块中仅提供了对数据的简单操作。

当完成操作后，应调用 cursor 的 close 方法关闭游标，然后调用 connection 对象的 close 方法关闭数据连接。如果执行了 SQL 查询语句，则可以使用 cursor 对象的 fetchall、fetchmany 或 fetchone 方法获取返回值。

下面所示的 odbc.py 脚本是使用 PythonWin 的 odbc 模块对数据进行操作。

```
# -*- coding:utf-8 -*-
# file: ODBC.py
#
import odbc                                          # 导入 odbc 模块
con = odbc.odbc('podbc')                             # 连接到数据库，即在数据源名中填写的名
字
cursor = con.cursor()                                # 创建 cursor 对象
cursor.execute('select id,name from people where id = 1') # 执行 SQL 语句查询 ID 为
1 的记录
r = cursor.fetchall()                                # 获得所有记录
print®                                               # 输出记录
cursor.execute('insert into people (name,age,sex)
                values (\'Jee\',21,\'female\')')# 添加记录
cursor.execute('DELETE FROM people where id = 3')  # 删除 ID 为 3 的记录
con.commit()                                         # 提交事务
cursor.close()                                       # 关闭 cursor
con.close()                                          # 关闭连接
```

16.1.2 使用 DAO 连接 Access 数据库

使用 ODBC 操作 Access 数据库时需设置数据源，如果数据库的路径被更改，则相应的 ODBC 数据源的配置也需要修改，这就使得使用 ODBC 连接数据过于烦琐。对于一些简单的数据应用情况，可以使用 DAO（Data Access Objects）代替 ODBC 连接数据库。

在 Python 中，使用 DAO 需要使用 PythonWin 提供的 win32com 对象，通过 win32com 对象来使用 Windows 的 COM 组件。首先，使用以下语句连接到 DAO 的 COM 对象上。

```
dbEngine = win32com.client.Dispatch('DAO.DBEngine.35')
```

然后通过 dbEngine 的 OpenDatabase 方法打开要连接的数据库。OpenDatabase 方法返回一个数据连接，然后就可以使用 OpenRecordset 方法打开数据库中的表。OpenRecordset 方法返回一个 Recordset 对象，使用 Recordset 可以对数据库进行操作。Recordset 对象由 Field 对象组成，Field 对象表示 Access 数据库中的一列。Recordset 对象的 Fields 表示了其所有 Field 对象的集合。

Recordset 对象常用的方法有以下几种。

- AddNew　　添加新记录。
- Close　　关闭 Recordset 对象。
- Delete　　删除记录。
- Find　　查找记录。
- Move　　移动位置。
- MoveFirst　　移动到第一条记录。

- MoveLast　　移动到最后一条记录。
- MoveNext　　移动到下一条记录。
- MovePrevious　　移动到上一条记录。
- Update　　更新记录。

如果 Access 数据库是使用较新版本的 Access 创建（如本章前面创建的数据库保存为*.accdb 格式），就不能使用 DAO 方式连接。因此，需要用 Access 2013 将其打开后重新保存为*.mdb 格式。

下面所示的 dao.py 脚本是使用 DAO 连接 Access 数据库。

```
# -*- coding:utf-8 -*-
# file: DAO.py
#
import win32com.client                                          # 导入win32com.client
dbEngine = win32com.client.Dispatch('DAO.DBEngine.36')          # 连接 COM 对象
daoDB = dbEngine.OpenDatabase('python.mdb')                     # 打开数据库
daoRS = daoDB.OpenRecordset('people')                           # 打开表
daoRS.MoveLast()                                                # 移动到最后一条记录
print(daoRS.RecordCount)                                        # 输出记录总数
print(daoRS.Fields('name').Value)                               # 输出最后一条记录的
name
print(daoRS.Fields('age').Value)                                # 输出最后一条记录的 age
print(daoRS.Fields('sex').Value)                                # 输出最后一条记录的 sex
daoRS.AddNew()                                                  # 添加新记录
daoRS.Fields('name').Value = 'Kate'                             # 新记录的 name
daoRS.Fields('age').Value = 22                                  # 新记录的 age
daoRS.Fields('sex').Value = 'Female'                            # 新记录的 sex
daoRS.Update()                                                  # 更新记录
daoRS.Close()                                                   # 关闭表
daoDB.Close()                                                   # 关闭数据库连接
```



在使用 win32com.client.Dispatch('DAO.DBEngine.36') 设置 DAO 引擎时，最好先查看一下当前计算机中用的是什么版本。具体的查看方法是，启动 PythonWin，选择菜单【Tools】|【COM Makepy utility】命令，打开如图 16-13 所示对话框。在对话框的列表中查找“Microsoft DAO”开头的项，后面显示的就是版本号。在图 16-13 中，可以看到其版本号是“Microsoft DAO 3.6”，因此在程序中应使用“DAO.DBEngine.36”，如果看到的是“Microsoft DAO 3.5”，则在程序中应使用“DAO.DBEngine.35”。



图 16-13　Select Library

16.1.3 使用 ADO 连接 Access 数据库

ADO（ActiveX Data Objects）是另一种简单的数据访问接口。使用 ADO 可以更快地创建连接

数据库的应用程序，ADO 具有更快的速度。ADO 和 DAO 相比有很多相似之处，但使用 ADO 也需要设置数据源。ADO 中也有 Recordset 对象，Recordset 对象常用的方法如下。

- AddNew　　添加新记录。
- Close　　关闭 Recordset 对象。
- Delete　　删除记录。
- Find　　查找记录。
- Move　　移动位置。
- MoveFirst　　移动到第一条记录。
- MoveLast　　移动到最后一条记录。
- MoveNext　　移动到下一条记录。
- MovePrevious　　移动到上一条记录。
- Update　　更新记录。

下面所示的 ado.py 脚本是使用 ADO 连接 Access 数据库。

```
# -*- coding:utf-8 -*-
# file: ADO.py
#
import win32com.client                                          # 导入 win32com.client
adoCon = win32com.client.Dispatch('ADODB.Connection')           # 创建连接对象
adoCon.Open('podbc')                                            # 连接到数据源
adoRS = win32com.client.Dispatch('ADODB.Recordset')             # 创建 Recordset 对象
adoRS.Open('[' + 'people' + ']', adoCon, 1, 3)                  # 打开数据源中的people表
adoRS.MoveFirst()                                               # 移动到第一条记录
for i in range(adoRS.RecordCount):
    print(adoRS.Fields('name').Value)                           # 输出记录的 name
    print(adoRS.Fields('age').Value)                            # 输出记录的 age
    print(adoRS.Fields('sex').Value)                            # 输出记录的 sex
    adoRS.MoveNext()
adoRS.AddNew()                                                  # 添加新记录
adoRS.Fields('name').Value = 'Kate'                             # 新记录的 name
adoRS.Fields('age').Value = 22                                  # 新记录的 age
adoRS.Fields('sex').Value = 'Female'                            # 新记录的 sex
adoRS.Update()                                                  # 更新记录
adoRS.Close()                                                   # 关闭表
adoCon.Close()                                                  # 关闭数据库连接
```

16.2 使用 MySQL 数据库

MySQL 是一个小巧的多用户、多线程 SQL 数据库服务器，已经得到了很多用户的认可，而且正在被更多的用户所使用。MySQL 是以客户机/服务器结构来实现的，它由一个服务器守护进程和客户程序组成。在 Python 中，可以使用 MySQLdb 模块连接到 MySQL 数据库，对 MySQL 数据库进行操作。

16.2.1 安装MySQL

MySQL的官方网站http://www.mysql.org/提供了Windows版的MySQL安装程序。MySQL的安装步骤较多，下面以mysql-5.1.72-win32.msi为例，介绍其安装过程，具体操作步骤如下。

step 1 从MySQL官方网站下载Windows版的MySQL安装程序mysql-5.1.72-win32.msi，下载完成后双击该安装程序，将显示如图16-14所示的安装向导界面。

step 2 单击【Next】按钮，进入安装形式选择界面，如图16-15所示。其中"Typical"项为典型安装，将MySQL安装到C盘；"Complete"项为完整安装，将安装所有功能；"Custom"项为自定义安装，可以选择安装路径和所安装的功能。



图16-14 MySQL安装向导界面

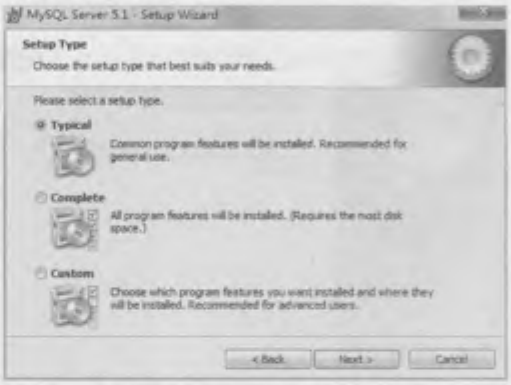


图16-15 安装形式选择界面

step 3 如果需要重新选择安装路径，则需选中【Custom】单选按钮，单击【Next】按钮，进入下一步安装，如图16-16所示。单击【Change】按钮可更改安装路径。

step 4 单击【Next】按钮后将出现如图16-17所示的确认安装界面，单击【Install】进行安装。

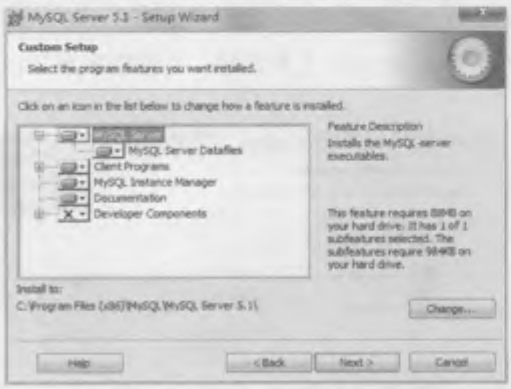


图16-16 选择路径和功能界面



图16-17 确认安装界面

step 5 文件复制完成后，将出现如图16-18所示的对MySQL企业版的一些介绍，可以单击几次【Next】按钮后进入如图16-19所示的【安装完成】界面。

step 6 单击【Finish】按钮完成安装，进入【数据库设置】界面，如图16-20所示。

step 7 单击【Next】按钮，进入【设置类型】界面，由于是初次安装，因此选中【Standard Configuration】单选框，如图16-21所示。



图 16-18 MySQL 企业版介绍界面



图 16-19 【安装完成】界面



图 16-20 【数据库设置】界面



图 16-21 【设置类型】界面

step 8 单击【Next】按钮，选中【Include Bin Directory in Windows PATH】选项，将 MySQL 添加到 PATH 环境变量，如图 16-22 所示。

step 9 单击【Next】按钮进入下一步安装，在密码框中填写登录到 MySQL 的密码，该密码将在连接数据库时使用。选中【Create An Anonymous Account】复选框，创建匿名用户，如图 16-23 所示。



图 16-22 设置选项



图 16-23 设置密码

step 10 单击【Next】进入下一步安装，显示如图 16-24 所示的【设置生效】界面，单击【Execute】按钮，完成安装。



图 16-24 【设置生效】界面

16.2.2 连接到 MySQL

安装好 MySQL 后，可以使用其附带的命令行工具创建数据库，由于是命令行工具，因此在 MySQL 中创建数据库不如使用 Access 直观(当然,也可以下载安装 MySQL 客户端工具,如 Navicat for MySQL 就是一个操作 MySQL 的图形化工具)。创建好数据库后，就可以使用 MySQLdb 模块，在 Python 中连接到数据，对其记录进行操作了。

1. 创建数据库

单击【开始】|【MySQL】|【MySQL Server 5.1】|【MySQL Command Line Client】命令，将运行 MySQL 的命令行管理工具。程序运行后会提示输入密码，此时输入安装 MySQL 时所设置的密码，按回车键后将出现下面所示的提示。

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.0.37-community-nt MySQL Community Edition (GPL)
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql>
```

在“mysql>”提示符下输入以下命令，创建一个名为“python”的数据库（斜体部分为用户输入部分)。

```
mysql> CREATE DATABASE python;
Query OK, 1 row affected (0.01 sec)
```

输入以下命令，使用刚刚创建的 python 数据库。

```
mysql> USE python;
Database changed
```

输入以下命令，创建一个名为 people 的表，people 表中包含 name、age 和 sex 项。

```
mysql> CREATE TABLE people (name VARCHAR(30), age INT, sex CHAR(1));
Query OK, 0 rows affected (0.20 sec)
```

输入以下命令，向表中添加记录，其中 NULL 表示项为空。

```
mysql> INSERT INTO people VALUES('Tom',20,'M');
```

```
Query OK, 1 row affected (0.06 sec)

mysql> INSERT INTO people VALUES('Jack',NULL,NULL);
Query OK, 1 row affected (0.06 sec)
```

输入以下命令，查看所创建的表中的内容。

```
mysql> SELECT * FROM people;
+------+------+------+
| name | age  | sex  |
+------+------+------+
| Tom  |   20 | M    |
| Jack | NULL | NULL |
+------+------+------+
2 rows in set (0.01 sec)
```

完成数据库创建后，可以使用“exit”命令退出 MySQL 的命令行窗口。

2. 安装 MySQLdb

在 Python 中，使用 MySQL 数据库需要安装 MySQLdb 模块。可以从其官方网站 http://sourceforge.net/projects/mysql-python/下载 Windows 下的安装程序。下面以 Python 3.2.5 为例，演示其具体安装步骤。

step 1 下载 MySQL-python-1.2.3.win32-py3.2.exe 文件，下载完成后双击该文件，如图 16-25 所示。

step 2 单击【下一步】按钮，如图 16-26 所示，安装程序将自动搜索安装的 Python。

step 3 单击【下一步】按钮，完成安装。



图 16-25 安装 MySQLdb



图 16-26 自动搜索 Python 的安装路径

当完成安装后，可以在交互式 Python Shell 输入以下所示语句。

```
import MySQLdb
```

如果上述语句成功运行，则说明 MySQLdb 已经安装成功。

3. 在 Python 中使用 MySQL 数据库

使用 MySQLdb 连接到 MySQL 数据库和使用 ODBC 连接到 Access 数据库类似。首先使用 MySQLdb 模块的 connect 方法连接到 MySQL 守护进程，connect 方法将返回一个数据库连接，使用数据库连接的 cursor 方法可以获得当前数据库的游标，然后就可以使用游标的 Execute 方法执行 SQL 语句，完成对数据库的操作。同样，当完成操作后，应调用 close 方法关闭游标和数据库

连接。

下面所示的PyMySQL.py脚本是使用MySQLdb连接到所创建的python数据库。

```
# -*- coding:utf-8 -*-
# file: PyMySQL.py
#
import MySQLdb                                          # 导入MySQLdb模块
db = MySQLdb.connect(host='localhost',                  # 连接到数据库，服务器为本机
        user='root',                                    # 用户名为root
        passwd='python',                                # 密码为python
        db='python')                                    # 数据库名为python
cur = db.cursor()                                       # 获得数据库游标
cur.execute('insert into people (name,age,sex) values (\'Jee\',21,\'F\')')
                                                        # 执行SQL语句，添加记录
r = cur.execute('delete from people where age=20')      # 执行SQL语句，删除记录
con.commit()                                            # 提交事务
r = cur.execute('select * from people')                 # 执行SQL语句，获取记录
r = cur.fetchall()                                      # 获取数据
print(r)                                                # 输出数据
cur.close()                                             # 关闭游标
db.close()                                              # 关闭数据库连接
```

16.3 嵌入式数据库SQLite

SQLite是一款轻型的嵌入式数据库，相对于其他的庞大数据库软件来说，SQLite显得十分小巧。SQLite可以满足一般的简单数据库应用，Python也提供了对SQLite的支持。

使用SQLite既不需要像MySQL一样通过守护进程进行，也不需要安装像Access那么庞大的软件，而只需要运行一个可执行文件即可。

SQLite无须安装，从其官方网站http://www.sqlite.org/下载一个Windows版的可执行文件即可。该可执行文件可以用于创建数据库，并向其中添加内容。

将从官方下载的sqlite-shell-win32-x86-3080100.zip解压至某一目录，然后通过命令运行sqlite3.exe，具体如下。

```
sqlite3.exe python
```

上述命令行的参数“python”表示创建一个名为“python”的数据库。运行sqlite3.exe后，将出现如下所示的提示。

```
SQLite version 3.8.1 2013-10-17 12:57:35
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

在“sqlite>”命令提示符后输入以下命令，可以在“python”中创建一个名为“people”的表。

```
sqlite> CREATE TABLE peple (name VARCHAR(30), age INT, sex CHAR(1));
```

输入以下所示命令，将向 people 表中插入 2 条数据。

```
sqlite> INSERT INTO people VALUES ('Tom', 20, 'M');
sqlite> INSERT INTO people VALUES ('Jack', 21, 'M');
```

输入以下所示命令，即可查看 people 表中的内容。

```
sqlite> SELECT * FROM people;
Tom|20|M
Jack|21|M
```

完成对数据库的操作后，可以使用 ".exit" 命令退出 SQLite 的命令行。

创建好 SQLite 中的数据库 "python" 后，即可在 Python 中使用这个数据库了。在 Python 中，使用 SQLite 数据库和使用 ODBC 操作 Access 数据库，以及操作 MySQL 数据库的过程类似。首先导入 sqlite3 模块，由于 SQLite 不需要服务器，因此直接使用 connect 方法打开数据即可。connect 方法会返回一个数据库连接对象，使用其 cursor 方法可以获得一个游标，然后就可以对记录进行操作了。在完成操作后，应使用 close 方法关闭游标和数据库连接。

下面所示的 PySqlite.py 脚本是在 Python 中操作 SQLite 数据库，查询显示出该数据库中的数据。

```
# -*- coding:utf-8 -*-
# file: PySqlite.py
#
import sqlite3                                          # 导入 sqlite3 模块
con = sqlite3.connect('python')                         # 连接到数据库
cur = con.cursor()                                      # 获得数据库游标
cur.execute('insert into people (name,age,sex) values (\'Jee\',21,\'F\')')
                                                        # 执行 SQL 语句，添加记录
r = cur.execute('delete from people where age=20')      # 执行 SQL 语句，删除记录
con.commit()                                            # 提交事务
cur.execute('select * from people')                     # 执行 SQL 语句，获取记录
s = cur.fetchall()                                      # 获得数据
print(s)                                                # 输出数据
cur.close()                                             # 关闭游标
con.close()                                             # 关闭数据库连接
```

16.4 本章小结

本章介绍了 Python 操作数据库方面的内容，主要介绍了连接、访问常用数据库系统的方法。首先介绍了使用三种方式（ODBC、DAO、ADO）连接访问 Access 数据库的操作，接着介绍了使用 Python 脚本操作 MySQL 数据库的方法，在这部分内容中还介绍了 MySQL 数据库的下载和安装。最后，介绍了使用 Python 操作嵌入式数据 SQLite 的方法。学会 Python 操作数据库的方法之后，就可以在 Python 中保存大量数据，并进行处理了。

下一章将进入另一个主题：Python 在 Web 方面的应用。

◆ ◆ ◆

第 17 章 Python Web 应用

本章包括

- 安装和使用 Web 应用服务器 Zope
- 在 Microsoft IIS 中使用 Python
- 使用 Plone 内容管理系统
- 在 Apache 中使用 Python

Python 可以和 ASP、PHP 等一样应用于 Web 服务。Python 还可以像 VBscript、Javascript 一样作为脚本嵌入到 ASP 中。Windows IIS 支持 ".py" 文件，可以使用 ".py" 文件代替 ".asp" 文件。除了在 Windows IIS 下使用 Python 外，在 Apache 中也可以使用 Python。除了使用 IIS 和 Apache 这类 Web 服务器程序外，还可以使用 Python 编写的 Web 服务器 Zope 来提供 Web 服务。

目前，已经有很多基于 Python 的 Web 框架技术，如 Plone、Django、TurboGears 等。使用这些 Web 框架可以大大简化 Web 应用程序的设计开发。

17.1 开源 Web 应用服务器 Zope

Zope 是一个开源 Web 应用服务器，其主要使用 Python 编写。Zope 使用了面向对象的思想，在 Zope 里一切都被称之为对象。Zope 提供了完善的功能，可以实现 ASP、PHP 和 JSP 的功能，并构建各种类型的 Web 应用。在 Zope 里，还可以使用数据库、模板等创建网站必不可少的功能。

17.1.1 安装 Zope

Zope 可以运行在多个操作系统下，在 Windows 下，可以从其官方网站 http://www.zope.org/ 下载 Windows 版的安装程序。对于初学者来说，下载 Windows 下的可执行安装程序进行安装是最好的方法，不过，现在 Zope 网站上主要提供源码下载，要下载安装程序，可通过 Zope 网站右下角提供的网站旧版链接 http://old.zope.org，在这里可下载到 Zope-2.11.4-win32.exe 安装程序，下面就以这个安装程序为例，介绍具体的安装过程。

step 1 双击运行 Zope-2.11.4-win32.exe，将显示如图 17-1 所示的安装向导界面。

step 2 单击【Next】按钮进入安装路径选择界面，此处可以根据需要重新设置 Zope 的安装路径，如图 17-2 所示。

step 3 单击【Next】按钮，进入【安装内容选择】界面，此处保持默认设置，如图 17-3 所示。

step 4 单击【Next】按钮，进入设置服务界面，此处保持默认设置，将 Zope 设置为 Windows 的服务，自动启动，如图 17-4 所示。如果未将 Zope 设置为 Windows 的服务，则每次使用 Zope 时需要手动启动。



图 17-1 Zope【安装向导】界面

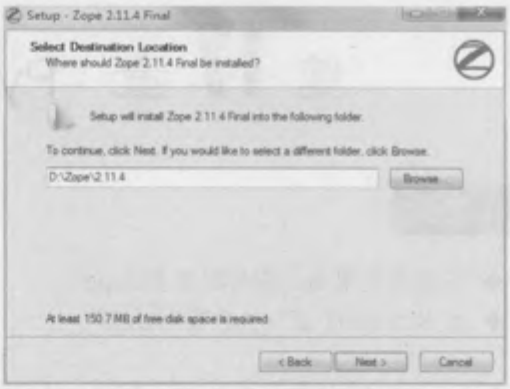


图 17-2 【安装路径选择】界面

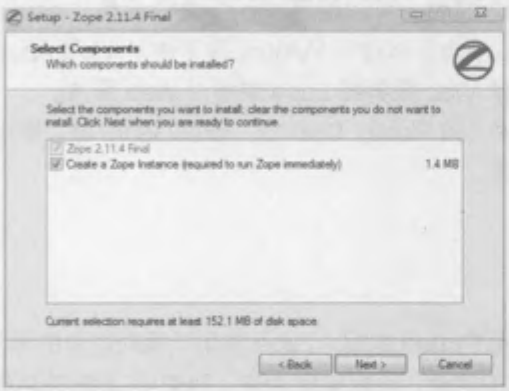


图 17-3 安装内容选择界面



图 17-4 将 Zope 设置为 Windows 的服务

step 5 单击【Next】按钮，进入【服务安装路径选择】界面，此处可以根据需要修改，如图 17-5 所示。

step 6 单击【Next】按钮，进入密码设置界面，该密码为登录 Zope 的管理密码，如图 17-6 所示。

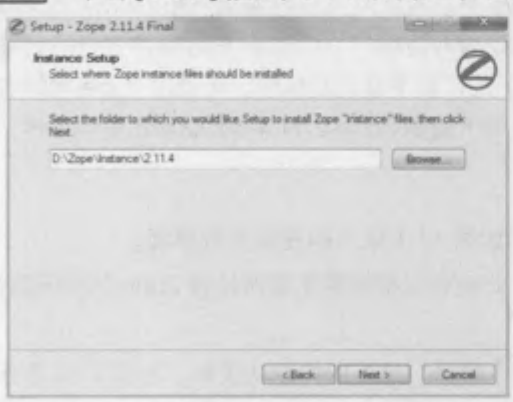


图 17-5 【服务安装路径选择】界面

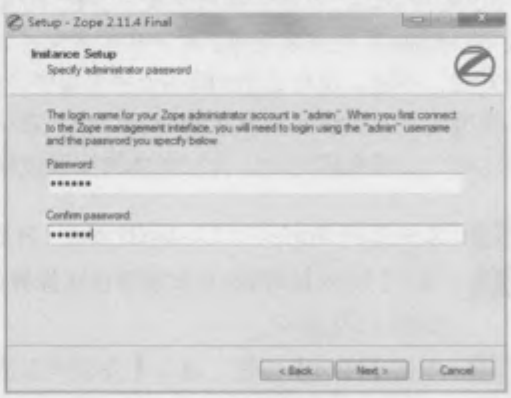


图 17-6 设置管理密码

step 7 单击【Next】按钮，进入【确认安装】界面，如图 17-7 所示。

step 8 单击【Install】按钮，进行安装，安装完成后将显示如图 17-8 所示界面，单击【Finish】按钮，完成安装。



图 17-7 【确认安装】界面

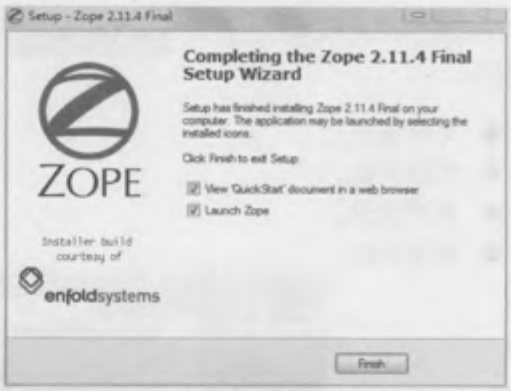


图 17-8 【安装完成】界面

当 Zope 安装完成后，可以在 IE 地址栏中输入 http://127.0.0.1:8080/或者 http://localhost:8080/，打开如图 17-9 所示的网页。如果未能打开页面，则可能是 Zope 服务没有启动，或者 Zope 安装失败。另外，如果当前计算机中安装了其他 Web 服务（如 IIS、Apache），并且这些 Web 服务占用了 8080 端口，则也无法打开如图 17-9 所示页面。



图 17-9 Zope 页面

17.1.2 使用 Zope 管理界面

安装好 Zope 后，在 IE 地址栏中输入 http://127.0.0.1:8080/manage，则可以打开 Zope 的管理界面。Zope 将要求输入密码，即安装 Zope 时所创建的密码，用户名为“admin”，如图 17-10 所示。

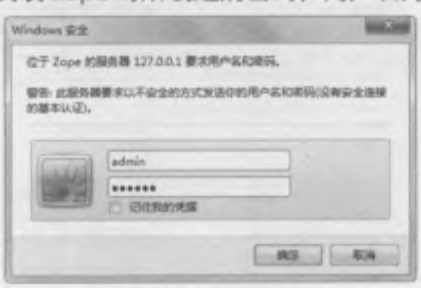


图 17-10 登录 Zope

单击【确定】按钮后，将进入 Zope 管理界面，如图 17-11 所示。Zope 管理界面由以下几部分组成。

- 对象管理。
- 控制面板。
- 用户管理。
- 错误日志。

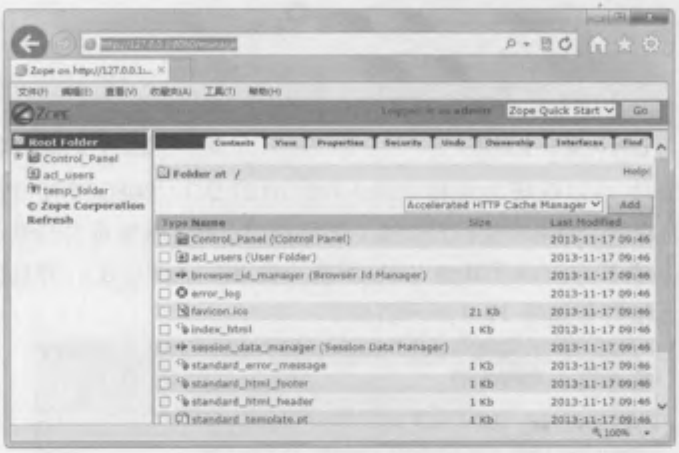


图 17-11　Zope 管理界面

对象管理，主要是添加、删除、修改对象、修改对象属性、权限等。对象管理是 Zope 最主要的功能，通过对象管理可以完成站点的建设。

控制面板，在控制面板中可以查看服务器的状态，关闭或者重新启动服务器。在控制面板中还可以设置数据库。

用户管理，可以用于创建新的 Zope 管理用户。

错误日志，记录了 Zope 所出现的错误。

下面首先通过 Zope 管理界面添加一个网页页面，具体操作步骤如下。

step 1　选择【Select type to add】下拉框中的【DTML Document】，如图 17-12 所示。

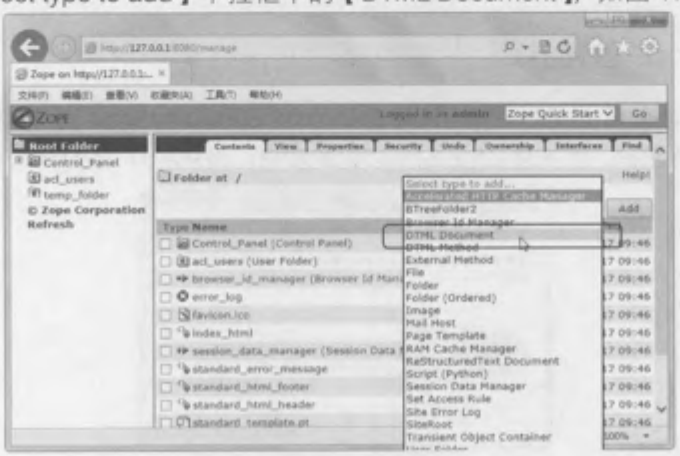


图 17-12　选择类型

step 2 单击【Add】按钮，将显示如图 17-13 所示界面，在【Id】文本框中填写“python”，在【Title】文本框中填写“Python”。

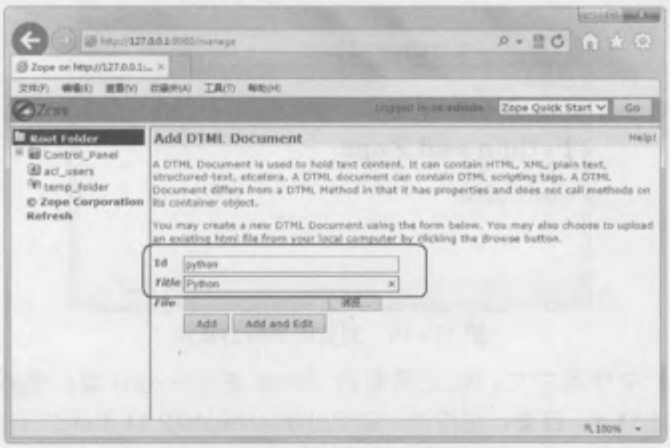


图 17-13 输入文件 Id 和 Title

step 3 单击【Add and Edit】按钮，将多行文本框中原有内容删除，输入以下所示内容，如图 17-14 所示。

```
<html>
<head>
<title>
Python
</title>
</head>
<body>
<h1>Python and Zope</h1>
<p>
<a href="http://www.python.org">Python Home Page</a>
</body>
</html>
```

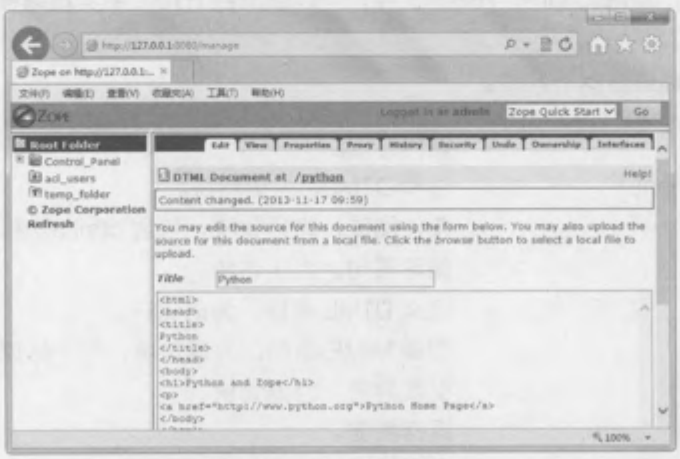


图 17-14 输入文件内容

step 4 单击【Save Changes】按钮，保存内容。

此时，可以通过填写的“Id”访问刚才创建的网页。在 IE 地址栏中输入 http://127.0.0.1:8080/python，将显示如图 17-15 所示的网页。



图 17-15 浏览所添加的文件

如果要在 Zope 中使用中文，则还需要对 Zope 进行一些设置。假设 Zope 服务安装在“D:\Zope\Instance\2.11.4”目录，则打开“D:\Zope\Instance\2.11.4\etc”目录中的“zope.conf”文件，将“default-zpublisher-encoding utf-8”前的注释去掉。如果没有“default-zpublisher-encoding utf-8”，则可以向“zope.conf”文件添加。包含中文的文件应该保存成“Utf-8”的格式，然后在图 17-13 所示界面中单击“File”浏览按钮，将文件上传到 Zope 中。

17.1.3 创建模板

在 Zope 里创建模板可以使用 DTML（Document Template Markup Language）和 ZPT（Zope Page Template）。DTML 是 Zope 中较早的模板语言，适合邮件模板、CSS 等的编写，而 ZPT 提供了创建页面模板更为有效的方式。

1. 使用 DTML

DTML 是由 DTML 命令列组成的，DTML 命令列是以“dtml-”为开头的标签。DTML 命令列可以和 HTML 混合在一起使用。使用 DTML 可以重复使用内容、保证内容格式统一和充分利用资源。DTML 的标签类似于 Javascript 和 VBscript 等，可以嵌入到 HTML 中实现编程。不同的是，DTML 是在服务器端执行的。

常用的 DTML 标签有以下几种。

- dtml-call　　调用方法。
- dtml-comment　　注释 DTML，为语句块。
- dtml-if　　条件测试，为语句块，包含 dtml-elif 和 dtml-else。
- dtml-in　　循环语句，为语句块。
- dtml-let　　定义 DTML 变量，为语句块。
- dtml-mime　　创建 MIME 编码，为语句块，用于处理邮件格式。
- dtml-raise　　引发异常，为语句块。
- dtml-return　　返回数据。
- dtml-sendmail　　发送邮件，为语句块。
- dtml-sqlgroup　　处理 SQL 语句，为语句块。
- dtml-sqltest　　测试 SQL 代码中的变量值。

- dtml-sqlvar　　向 SQL 代码中插入变量。
- dtml-tree　　创建树组件，为语句块。
- dtml-try　　捕捉异常，为语句块，包含 dtml-except 和 dtml-finally。
- dtml-unless　　条件测试，为语句块。
- dtml-var　　插入变量。
- dtml-with　　在命名空间中查找变量，为语句块。

除了上述标签外，DTML 还提供了内置函数，用于完成简单的任务。DTML 标签可以和 HTML 标签混合使用。

下面所示的步骤是使用 DTML 在 Zope 中创建一个简单的模板。

step 1 登录 Zope 管理界面，添加一个 Id 为“head”的 DTML Document 文件，其内容如下。

```
<table border="1">
  <tr>
    <td><font size="10">Python Web</font></td>
  </tr>
</table>
```

step 2 添加一个 Id 为“foot”的 DTML Document 文件，其内容如下。

```
<table  border="1">
  <tr>
    <td align="right">Powered by Zope</td>
  </tr>
</table>
```

step 3 添加一个 Id 为“dpt”的 DTML Document 文件，其内容如下。

```
<html>
<dtml-var head>
<h2><dtml-var title_or_id></h2>
<p>
This is the <dtml-var id> Document.
</p>
<dtml-var foot>
</html>
```

在“dpt”中，主要使用 dtml-var 标签，在其头部插入所创建的“head”文件，在其结尾插入所创建的“foot”文件。保存文件后，在 IE 地址栏中输入 http://127.0.0.1:8080/dpt，将显示如图 17-16 所示的内容，可以看到，在这个网页上方显示了前面定义的 id 为“head”中的内容，而下方显示了前面定义的 id 为“foot”中的内容。



图 17-16　使用 DTML 创建的模板

2. 使用 ZPT

在 ZPT 中，主要使用 TAL（Template Attribute Language）定义模板。TAL 是由以“tal:”开头

的一系列标记、属性以及其他相关的值组成。TAL 既可以嵌套在 HTML 中，也可以嵌套在 XML 中。

使用 ZPT 创建模板可以使用 Dreamweaver 这样的所见即所得的编辑器创建模板，然后可以向模板中添加 TAL，添加了 TAL 的模板仍可以被 Dreamweaver 修改而不受影响。

ZAL 常用的标记有以下几种。

- tal:attributes　　修改属性。
- tal:condition　　条件测试。
- tal:content　　输出内容。
- tal:define　　定义变量。
- tal:on-error　　异常处理。
- tal:repeat　　循环。
- tal:replace　　替换内容。

在 Zope 中添加一个 Id 为“zpt”的 Page Template 模板（在图 17-12 所示的下拉列表中选择【Page Template】），设置其内容如下。

```
<html>
  <head>
    <title tal:content="template/title">The title</title>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
  </head>
  <body>
<table border="1" width="100%">
  <tr>
    <th>Id</th>
  </tr>
  <tr tal:repeat="item context/objectValues">
    <td tal:content="item/getId">Id</td>
  </tr>
</table>
  </body>
</html>
```

保存“zpt”后，在 IE 地址栏中输入 http://127.0.0.1:8080/zpt，将显示如图 17-17 所示的页面，可以看到，通过 tal:repeat 标记循环生成了表格的多行数据。



图 17-17　使用 ZPT 创建的模板

17.1.4 添加 Python 脚本

在 Zope 中使用的 Python 脚本和普通的 Python 脚本并没有什么区别，但为了提高安全性，Zope 对 Python 脚本做了一些限制。例如，不能使用内置 open 函数、range 函数不能产生较大的循环等。

在 Zope 中，表单所提交的内容也可以通过参

数的形式传递给 Python 脚本。

在 Zope 管理页面，添加一个 Id 为 “submit” 的 DTML Document 文件，设置其内容如下。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Input your name and age</title>
</head>

<body>
<form id="form1" name="form1" method="post" action="show">
  <label>Name
  <input type="text" name="name" />
  </label>
  <p>
    <label>Age
    <input type="text" name="age" />
    </label>
  </p>
  <p>
    <input type="submit" name="Submit" value="提交" />
    <input type="reset" name="Submit2" value="重置" />
  </p>
</form>
</body>
</html>
```

接着，在 Zope 管理页面中再添加一个 Id 为 “show” 的 Script(Python)脚本，在其【Parameter List】文本框中输入 “name,age”，即表单所提交的变量名。修改其内容如下。

```
print("<center>")
print("<p>Your name is:</p><p>")
print(name)
print("</p><p>Your age is:</p><p>")
print(age)
print("</p></center>")
return(printed)
```

在 IE 地址栏中输入 http://127.0.0.1:8080/submit，在【Name】文本框中输入 “Tom”，在【Age】文本框中输入 “20”，如图 17-18 所示。单击【提交】按钮后，将显示如图 17-19 所示的结果。



图 17-18 输入【Name】和【Age】

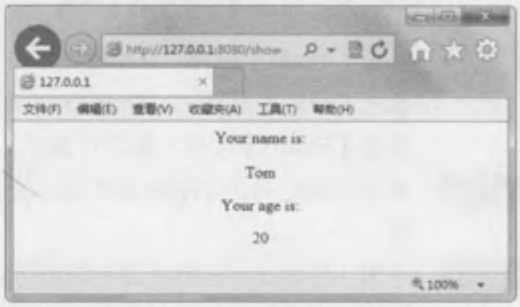


图 17-19 Python 脚本输出

17.2 使用 Plone 内容管理系统

Zope 的使用过于复杂，对于初学者而言，在 Zope 下完整地建立一个网站需要花费的时间较多，而使用 Plone 建立网站则较为容易。

Plone 是基于 Zope 的内容管理系统，使用 Plone 可以建立网站、内容发布系统等。在企业内部，Plone 可以作为企业内部网的服务器，发布通知、文档等。Plone 还提供了 Blog、论坛、Wiki 等扩展产品，为用户提供了丰富的选择。

17.2.1 安装 Plone

在 Plone 官方网站 http://plone.org 提供了 Windows 下的完整安装程序，其中包含了 Zope。该安装程序适合初学者使用，避免了复杂的配置。如果已经安装了 Zope，则可以在 Zope 下安装 Plone。为了方便，最好卸载 Zope，然后重新安装带有 Zope 的 Plone。

1. 安装 Plone

下面以 Plone-4.3.1-win32.exe 的安装为例，介绍 Plone 在 Windows 下的安装过程，具体操作步骤如下。

step 1 双击运行安装程序 Plone-4.3.1-win32.exe，打开如图 17-20 所示的【安装向导】界面。

step 2 单击【Next】按钮，进入如图 17-21 所示的【准备安装】界面。



图 17-20 Plone【安装向导】界面



图 17-21 【准备安装】界面

step 3 单击【Install】按钮，开始复制文件，在这个版本中没有让用户设置安装位置的界面，直接将 Plone 安装到 C:\Plone43 目录下。完装完成后将显示如图 17-22 所示的提示界面，单击【Finish】按钮，重启计算机。

step 4 重启计算机后，打开 IE 浏览器，输入 http://127.0.0.1:8080/，将显示如图 17-23 所示的界面。

step 5 从图 17-23 所示的网页显示内容可以看出，Plone 已经在运行。单击【Zope 管理界面】链接，将显示如图 17-24 所示的界面，输入管理员账号和密码（初始值都为 admin）。

step 6 单击【确定】按钮后，将显示如图 17-25 所示的 Zope 管理界面，与上节介绍的单独安装的 Zope 管理界面相似。



图 17-22 【安装完成】界面



图 17-23 打开的 Plone 界面



图 17-24 输入管理页账号和密码

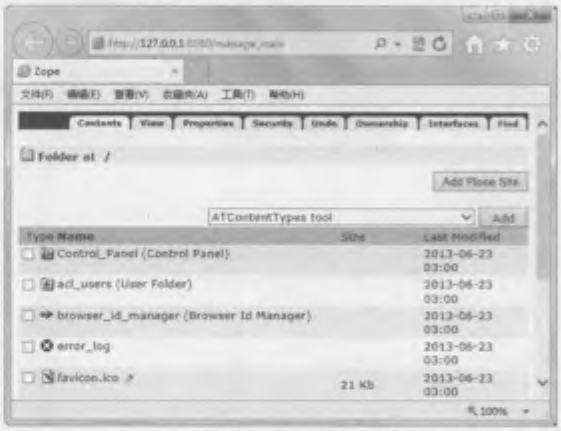


图 17-25 Zope 管理界面

2. 新建 Plone 站点

刚安装完的 Plone，还没有一个站点，因此将看到图 17-23 中的网页信息，接下来就可以创建站点了。

step 1 在图 17-23 所示界面中单击【创建一个新 Plone 站点】按钮，将显示如图 17-26 所示的界面，输入新建站点的各种信息。

step 2 单击图 17-26 所示页面最下方的【创建 Plone 站点】按钮，创建新的站点，创建完成后将显示新建站点的页面，如图 17-27 所示。

创建站点时，是以 admin 账号进行操作的，因此站点中显示的用户名仍然是 admin，如图 17-27 右上角所示。通过这个账号还可以进行站点的管理操作。

step 3 单击图 17-27 右上角的 admin 账号，将显示一个下拉菜单，从中选择【网站设置】命令，将显示如图 17-28 所示的【网站设置】界面，可以看出，通过这个页面可以对网站进行各项设置，如设置导航、设置网站安全、管理用户等。

step 4 用户也可以很方便地发布、编辑网站中的新闻信息。例如，在图 17-27 所示界面中显示了一条“欢迎使用 Plone 系统”的页面信息，单击上方导航栏中的【编辑】链接，将显示如图 17-29 所示的编辑页面，滚动页面可查看、编辑页面的标题、摘要、正文等相关信息。



图 17-26　输入新建站点的各种信息



图 17-27　新建站点的页面



图 17-28　【网站设置】界面



图 17-29　编辑页面

17.2.2　安装 Plone 插件

除了基本的内容管理系统外，Plone 还以插件的形式提供了很多产品，这些产品也可以被安装到 Plone 中。下面以 Plone 官方网站提供的 ftw.blog 为例（ftw.blog 是在 Plone 中创建 Blog 的产品），演示在 Plone 中安装产品的步骤，具体操作如下。

step 1　从 Plone 的官方网站查找 Weblogs 插件，将显示如图 17-30 所示的结果。

step 2　单击 ftw.blog 插件，将显示该插件的相关介绍，注意其“Install”部分的内容，如图 17-31 所示，将这部分内容记下来，配置安装时会用到。

step 3　用记事本打开 C:\Plone43 目录中的文件“buildout.cfg”，其中的内容如下，在最后添加图 17-31 中所看到的文本“ftw.blog”。

```
eggs =
```

```
Plone
Pillow
Products.PloneHotfix20130618
ftw.blog
```



图 17-30 查找 Weblogs 插件



图 17-31 查看 ftw.blog 插件的信息

step 4 进入 Windows 的命令窗口，切换到 C:\Plone43 目录，执行以下命令：

```
C:\Plone43> .\bin\buildout.exe
```

这时，buildout 命令将查找插件 ftw.blog 并下载该插件的相关文件（应保证计算机能连接上互联网）。经过一段时间的下载、编译后，完成插件的安装。



在插件下载安装过程中，由于文件“buildout.cfg”中的“eggs”部分还有其他插件，而有一些插件可能无法下载，因此会出现一些错误提示，但这并不影响 ftw.blog 的下载和编译，只要最后看到有类似于下面的提示信息，就表示 ftw.blog 下载安装成功。

```
*************** PICKED VERSIONS ****************
[versions]
ftw.blog = 1.5
ftw.colorbox = 1.1.4
ftw.tagging = 1.1.0
ftw.upgrade = 1.7.0
products.addremovewidget = 1.5.1

*************** /PICKED VERSIONS ***************
```

step 5 在 IE 浏览器中打开如图 17-25 所示的 Zope 管理界面，单击【Control_Panel (Control Panel)】进入控制面板，如图 17-32 所示。

step 6 单击下方的【Restart】按钮，重启 Zope。

step 7 重新启动 Zope 服务后，登录 Plone 界面，单击网页右上角的账号“admin”，从下拉菜单中选择【网站设置】命令，打开如图 17-28 所示【网站设置】界面，在这个页面单击【附加组件】链接，将显示如图 17-33 所示界面。



图 17-32　控制面板



图 17-33　附加组件

step 8　在图 17-33 所示页面中，单击勾选“ftw.blog 1.5”，然后单击下方的【启用】按钮，即可在站点中启用该插件。

step 9　重新回到站点首页，单击【添加新...】链接，下拉列表中将显示“Blog”项，如图 17-34 所示，单击选择“Blog”链接，将显示如图 17-35 所示的【添加 Blog】页面，在这里输入 Blog 的内容后即可发布。



图 17-34　单击【添加新...】链接



图 17-35　【添加 Blog】页面

17.3　在 Microsoft IIS 中使用 Python

Microsoft IIS 是 Microsoft 提供的 Web 服务器，在 IIS 中可以使用 ASP（Active Server Pages）创建动态网站。需要注意的是，ASP 本身并不是脚本语言，在 ASP 中可以嵌入其他的脚本语言，如 VBscript、Javascript 或 Python。其实，也可以直接在 IIS 中使用 Python 脚本代替“.asp”文件。

17.3.1 安装 Microsoft IIS

在 Windows 下，需要安装 Microsoft IIS 才能支持 ASP。默认情况下，安装 Windows 7 时不会安装 IIS 组件，因此，要想使用 ASP，则必须在 Windows 7 中安装 IIS，具体安装步骤如下。

step 1 单击【开始】|【控制面板】|【程序和功能】命令，打开如图 17-36 所示对话框。

step 2 在图 17-36 所示对话框中单击左侧的【打开或关闭 Windows 功能】，将打开如图 17-37 所示对话框，展开【Internet 信息服务】，根据需要选中相应的项，图 17-37 中选择了大部分的功能，包括 Web 管理工具、应用程序开发功能中的 ASP/ASP.NET 等。

step 3 单击【确定】按钮，将显示如图 17-38 所示的【提示】窗口，Windows 7 开始根据设置更改功能，经过几分钟时间，更改完成后将自动关闭图 17-37 所示对话框。

step 4 安装完成后，在 IE 地址栏中输入 http://localhost/，将打开如图 17-39 所示的界面，表示 IIS 已经安装成功。



图 17-36 【卸载/或更改程序】对话框



图 17-37 选择需要打开的功能

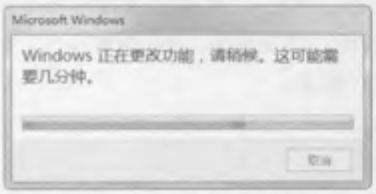


图 17-38 【提示】窗口



图 17-39 IIS 起始页面

step 5 IIS 安装完成后，单击【开始】|【管理工具】|【Internet 信息服务（IIS）管理器】命令，

将打开如图 17-40 所示窗口。

step 6 单击窗口左侧列表（默认是计算机名称），将其展开，接着展开【网站】，单击选中【Default Web Site】项，双击窗口中间的【IIS】部分，找到【处理程序映射】命令，将显示如图 17-41 所示的对话框。



图 17-40 IIS 管理界面



图 17-41 【处理程序映射】对话框

step 7 在图 17-41 所示窗口中，单击右侧的【添加脚本映射】命令，将打开如图 17-42 所示对话框，在【请求路径】中填写 Python 程序的扩展名“*.py”，在【可执行文件】中填写 Python 的执行程序，在【名称】中输入一个标识名称，这里输入“python”。

step 8 单击【确定】按钮，即可设置 IIS，使其支持 Python 脚本。



图 17-42 【添加脚本映射】对话框

17.3.2 在 ASP 中使用 Python 脚本

在“.asp”文件中可以像嵌入 VBscript 脚本一样嵌入 Python 脚本，也可以在 ASP 中直接使用 Python 脚本。在 ASP 中使用 Python 脚本基本上没有什么限制，与普通的 Python 脚本编写没有太大的区别。

1. 在“.asp”文件中包含 Python 脚本

在“.asp”文件中包含 Python 脚本，需要在文件开头使用“<%@LANGUAGE=Python%>”指明。然后就可以在“<%”和“%>”之间使用 Python 脚本。需要注意的是，在“<%”和“%>”之间使用 Python 脚本时仍要注意保持缩进。另外，Python 的 print 不能输出，需要使用 Response.Write 输入数字或者字符串。下面所示的 psptest.asp 脚本是在 ASP 中使用 Python。

```
<%@LANGUAGE=Python%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>use Python in ASP</title>
</head>
<body>
<h1>use Python in ASP</h1>
<%
import os                                                    # 导入 os 模块
class Info:                                                  # 定义类
    def __init__(self):
        Response.Write('<h1>Python Class </h1>')
    def show(self):
        Response.Write('<h1>Class Info</h1>')
def print_br():                                              # 定义函数
    Response.Write('<br>')
def print_h1(s):
    Response.Write('<h1>')
    Response.Write(s)
    Response.Write('</h1>')
print_h1('使用 os 模块')                                      # 调用函数
for path in os.sys.path:                                     # 使用 os 模块
    Response.Write(path)
```

```
    print_br()
print_h1('使用 string 模块')
for s in str.split('Python is great'):                    # 使用 string 模块
    Response.Write(s)
    print_br()
print_h1('使用类')
info = Info()                                             # 类实例化
info.show()                                               # 调用类方法
%>
</body>
</html>
```

脚本中使用“<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />”表示为utf-8 编码， psptest.asp 应保存为 utf-8 格式。将 psptest.asp 保存至 IIS 默认网站的目录“C:\Inetpub\wwwroot”。在 IE 浏览器中输入 http://127.0.0.1/psptest.asp，将显示如图 17-43 所示效果。



图 17-43 在 IE 中打开 psptest.asp

2. 直接使用 Python 脚本

除了在 ASP 脚本文件中嵌入 Python 脚本之外，IIS 也可以直接解释 Python 脚本。下面所示的 pythonasp.py 脚本可以代替“.asp”文件。

```
# -*- coding:utf-8 -*-
# file: pythonasp.py
#
import os                                                 # 导入 os 模块
Print( '''
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Python</title>
</head>
<body>
```

```
''')
Print('<h1>Python 路径</h1>')
i = 1
for path in os.sys.path:                                    # 使用 os 模块
    print(i, ' ', path)
    print('<br>')
    i = i + 1
print('''
</body>
</html>
''')
```

将 pythonasp.py 保存为 utf-8 格式，将其保存至 IIS 默认网站的目录“C:\Inetpub\wwwroot”。在 IE 浏览器中输入 http://127.0.0.1/pythonasp.py，将显示如图 17-44 所示效果。

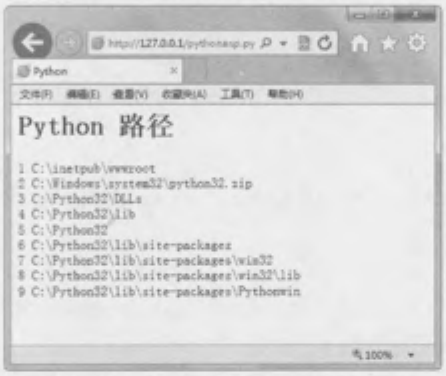


图 17-44 在 IE 中打开 pythonasp.py

17.3.3 一个简单的例子

从前面的例子可以看出，在 IIS 中使用 Python 和使用 Python 脚本没有什么区别。在 ASP 中通常会使用 Access 数据库来保存信息，而在使用了 Python 之后，则完全可以使用 SQLite 作为数据库。本节中将给出一个使用 SQLite 数据库保存信息的简单留言板的例子。

首先在 SQLite 创建一个存储留言的数据库，其步骤如下（斜体部分为用户输入命令）。

```
E:\Python\第 17 章>sqlite3.exe message
SQLite version 3.8.1 2013-10-17 12:57:35
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> CREATE TABLE message (name TEXT, mail TEXT, site TEXT, content TEXT, time
DATETIME);
sqlite> .exit
```

创建提交页面 submit.html，其内容如下。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<title>提交留言</title>
</head>

<body>
```

```
<table>
  <tr>
    <td>
     <h1>提交留言</h1>
     <br />
     <form id="form" name="form" method="post" action="addmessage.py">
       <label>姓名
       <input type="text" name="name" />
       </label>
       <p>
         <label>邮箱
         <input type="text" name="email" />
         </label>
       </p>
       <p>
         <label>网站
         <input type="text" name="site" />
         </label>
       </p>
       <p>
         <label>留言内容：<br />
         <textarea name="content" cols="50" rows="10"></textarea>
         </label>
       </p>
       <p>
         <input type="submit" name="Submit" value="提交" />
         <input name="Cancel" type="reset" id="Cancel" value="重置" />
       </p>
     </form>
     </td>
  </tr>
</table>
</body>
</html>
```

创建添加留言页面 addmessage.py（应保存为 utf-8 格式），其内容如下。

```
# -*- coding:utf-8 -*-
# file: addmessage.py
#
import cgi                                                    # 导入 cgi 模块处理表单数据
import sqlite3                                                # 导入 sqlite3 模块
import datetime
form = cgi.FieldStorage()                                     # 创建表单对象
name = unicode(form["name"].value, 'GBK')                     # 获得表单变量，并改变其编码
mail = unicode(form["email"].value, 'GBK')
site = unicode( form["site"].value, 'GBK')
content = unicode(form["content"].value, 'GBK')
now = datetime.datetime.now()                                 # 获得当前时间
time = now.strftime('%Y-%m-%d %H:%M:%S')                      # 格式化时间
con = sqlite3.connect('message')                              # 连接到数据库
cur = con.cursor()
cur.execute("INSERT INTO message VALUES(?,?,?,?,?)", (name, mail, site, content,
time))
con.commit()
```

```
cur.close()
con.close()
print('''
<html>
<head>
<title>添加成功</title>
</head>
<body>
<h1>添加成功</h1>
<br>
<a href=show.py>单击查看留言</a>
</body>
</html>
''')
```

创建查看留言页面 show.py（应保存为 utf-8 格式），其内容如下。

```
# -*- coding:utf-8 -*-
# file: PySqlite.py
#
import sqlite3                                          # 导入 sqlite3 模块
con = sqlite3.connect('message')                        # 连接到数据库
cur = con.cursor()                                      # 获得数据库游标
'GBK')
cur.execute('select * from message')                    # 执行 SQL 语句
results = cur.fetchall()                                # 获得数据
print('''
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>use Python in ASP</title>
</head>
<body>
<center>
<h1>所有留言</h1>
</center>
<hr />
''')
for result in results:
    print('姓名:',result[0].encode('UTF-8'))
    print('<br>')
    print('时间:',result[4].encode('UTF-8'))
    print('<br>')
    print('邮箱:', result[1].encode('UTF-8'))
    print('<br>')
    print('网站',result[2].encode('UTF-8'))
    print('<br>')
    print('留言内容:')
    print('<br>')
    print(result[3].encode('UTF-8'))
    print('<hr />')
print('''
</body>
</html>
''')
```

```
cur.close()                                    # 关闭游标
con.close()                                    # 关闭数据库连接
```

将上述页面和 message 数据库保存到 IIS 的网站目录，默认为“C:\Inetpub\wwwroot”，在 IE 地址栏中输入 http://127.0.0.1/submit.html，将显示如图 17-45 所示的界面。输入留言内容，单击【提交】按钮提交留言。重复操作，再添加几条留言信息。

添加留言之后，将显示一个“查看留言”链接，单击该链接将显示如图 17-46 所示的所有留言内容。

该留言板很简单，只做演示用，读者可以自行将其完善。



由于目录“C:\Inetpub\wwwroot”默认对网页浏览用户是没有写权限的，因此，运行以上脚本时会有一个错误提示，如图 17-47 所示。由于用户对数据库没有写权限，因此，错误提示数据库是只读的。这时，可将目录“C:\Inetpub\wwwroot”设置为 User 用户可写（或者直接设置 Everyone 用户可写），就不会出现如图 17-47 所示的错误了。



图 17-45　提交留言



图 17-46　查看留言



图 17-47　错误提示

17.4 在 Apache 中使用 Python

Apache 是安全性很高、非常流行的开源 HTTP 服务器，可以运行在多种操作系统中。在 Apache 中，可以使用 Perl、Python 和 PHP 等作为编程语言进行网站开发。在 Apache 中使用 Python 时，需要下载安装 mod_python 模块。

17.4.1 安装配置 Apache

Apache 提供了 Windows 下的安装程序。由于 Apache 使用文本文件作为配置文件，因此其配置过程较为烦琐。

1. 安装 Apache

Apache 的安装程序可以从其官方网站 http://www.apache.org/下载，以 apache_2.2.4-win32-x86-openssl-0.9.8d.msi 为例，其安装步骤如下。

step 1 双击运行安装程序，打开如图 17-48 所示的【安装向导】界面。

step 2 单击【Next】按钮，进入【安装协议】界面，如图 17-49 所示，选中【I accept the terms in the license agreement】单选项。



图 17-48 【安装向导】界面



图 17-49 【安装协议】界面

step 3 单击【Next】按钮，进入【安装说明】界面，如图 17-50 所示。

step 4 单击【Next】按钮，进入【服务器信息设置】界面，如图 17-51 所示，此处可以设置服务器的域名、主机名等。如果没有域名则可以不填写。



图 17-50 【安装说明】界面

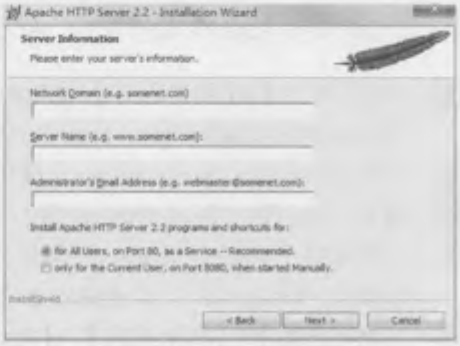


图 17-51 【服务器信息设置】界面

step 5 单击【Next】按钮，进入【安装类型选择】界面，如图 17-52 所示。

step 6 单击【Next】按钮，进入【安装路径选择】界面，此处可以根据需要设置 Apache 的安装路径，如图 17-53 所示。



图 17-52 【安装类型选择】界面



图 17-53 【安装路径选择】界面

step 7 单击【Next】按钮，进入【确认安装】界面，如图 17-54 所示，单击【Install】按钮安装 Apache。

step 8 安装完成后，安装程序会将 Apache 设为 Windows 服务，并启动 Apache。

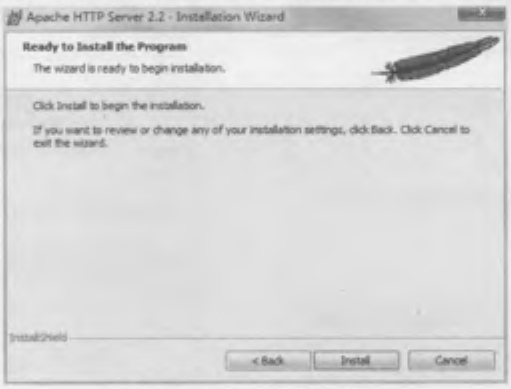


图 17-54 【安装确认】界面

2. 配置 Apache

如果已经安装了 Microsoft IIS，并且在安装过程中没有使用正确的域名，则 Apache 可能会出现错误。此时需要对 Apache 进行配置，Apache 的配置过程如下。

step 1 单击【开始】|【所有程序】|【Apache HTTP Server 2.2.4】|【Configure Apache Server】|【Edit the Apache httpd.conf Configuration File】命令，打开 Apache 配置文件。

step 2 将第 133 行的“ServerAdmin”使用“#”注释掉。

step 3 将第 142 行的“ServerName :80”使用“#”注释掉。

step 4 如果安装了 Microsoft IIS，则需要修改 Apache 监听的端口，将第 53 行的“Listen 80”修改为“Listen 82”。修改完以上项目之后保存该配置文件。

step 5 单击【开始】|【所有程序】|【Apache HTTP Server 2.2.4】|【Control Apache Server】|【Restart】命令，重新启动 Apache。

step 6 在 IE 地址栏中输入 http://127.0.0.1:82 或者 http://localhost:82/，将显示 "It works!"，表明 Apache 已经正常工作。

17.4.2 安装 mod_python

mod_python 是用于使 Apache 支持 Python 脚本的模块，使用 mod_python 可以在 Apache 中使用 PSP（Python Server Pages）或者使用 Python 编写 CGI。mod_python 可以从其官方网站 http://www.modpython.org/下载。

mod_python 不仅作为 Apache 的模块，也作为 Python 的模块安装到 Python 中。因此需要根据所安装的 Python 的版本，从其官方网站下载相应的 mod_python 安装文件。以 Python 3.2 为例，mod_python 的安装配置过程如下。

step 1 从 mod_python 官方网站下载 mod_python-3.3.1.win32-py2.5-Apache2.2.exe 安装程序，双击运行安装程序，打开如图 17-55 所示的【安装向导】界面。



由于现在没有提供能适用于更高版本 Python 的 mod_python 模块，这里为了演示 Python 在 Apache 中的使用，只能另外再安装一套 Python 2.5。

step 2 单击【下一步】按钮，进入【安装路径选择】界面，如图 17-56 所示。



图 17-55 【安装向导】界面



图 17-56 【安装路径选择】界面

step 3 单击【下一步】按钮，开始进行安装，在 mod_python 安装过程中将会弹出如图 17-57 所示的【设置 Apache 安装路径】的界面。



图 17-57 【设置 Apache 安装路径】界面

step 4 安装完成后，单击【开始】|【所有程序】|【Apache HTTP Server 2.2.4】|【Configure Apache

Server】|【Edit the Apache httpd.conf Configuration File】命令，打开 Apache 配置文件，在第 66 行以后添加如下所示语句，即可在 Apache 中载入 mod_python 模块。

```
LoadModule python_module modules/mod_python.so
```

step 5 在<Directory "D:/Apache Software Foundation/Apache2.2/htdocs">（根据 Apache 所安装的路径，此处可能会有所不同）和</Directory>之间添加如下所示语句。

```
AddHandler mod_python .py
PythonHandler pythontest
PythonDebug On
```

step 6 在"D:/Apache Software Foundation/Apache2.2/htdocs"中编写 pythontest.py，其内容如下。

```
# -*- coding:utf-8 -*-
# file: pythontest.py
#
from mod_python import apache
def handler(req):
    req.content_type = 'text/html'
    req.write('''
     <html>
     <head>
     <title>Python</title>
     </head>
     <body>
     <h1>mod_python<h1>
     </body>
     </html>
     ''')
return apache.OK
```

step 7 单击【开始】|【所有程序】|【Apache HTTP Server 2.2.4】|【Control Apache Server】|【Restart】命令，重新启动 Apache。

step 8 在 IE 地址栏中输入 http://127.0.0.1:82/pythontest.py 或者 http://localhost:82/pythontest.py，将显示“mod_python”，表明 mod_python 可以使用了。

17.4.3 使用 Python Sever Pages 创建留言板

在 17.4.2 节的测试例子中，在 Apache 配置文件中添加了“PythonHandler pythontest”语句，并添加了一个发布处理器。发布处理器是一个 Python 脚本，mod_python 使用该脚本来处理 Apache 发出的请求。配置文件中，“AddHandler mod_python .py”表示所有的“.py”文件都由 mod_python 处理。

如果觉得自己创建发布处理器较为烦琐，则可以使用 mod_python 的标准发布处理器，将“PythonHandler pythontest”修改为“PythonHandler mod_python.publisher”即可。

mod_python 还支持 PSP（Python Sever Pages），可以使用类似于 ASP 的语法嵌入 Python。为了使 Apache 支持 PSP，需要将“AddHandler mod_python .py”修改为“AddHandler mod_python .psp”，将“PythonHandler pythontest”修改为“PythonHandler mod_python.psp”。重新启动 Apache 后即可在 Apache 中使用 PSP 创建页面。

在本小节中，将修改上一节中的使用 SQLite 制作留言板的例子，将数据库修改为 MySQL，环境改为 Apache。为了让 MySQL 支持中文，首先应修改其配置文件 my.ini。my.ini 位于 MySQL 的安装目录中，例如，“D:\Program Files\MySQL\MySQL Server 5.0”。将“default-character-set”项值改为“gbk”，如下所示。

```
[mysql]
default-character-set=gbk
```

修改后重新启动 MySQL 服务。单击【开始】|【所有程序】|【MySQL】|【MySQL Server 5.0】|【MySQL Command Line Client】命令，输入如下命令，创建留言本的数据库（斜体部分为用户输入命令）。

```
Enter password: ******
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 17
Server version: 5.0.37-community-nt MySQL Community Edition (GPL)
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql>CREATE DATABASE message DEFAULT CHARACTER SET gbk COLLATE gbk_chinese_ci;
Query OK, 1 row affected (0.02 sec)
mysql> USE message;
Database changed
mysql> CREATE TABLE message(name TEXT, mail TEXT, site TEXT, content TEXT, time
DATETIME);
Query OK, 0 rows affected (0.13 sec)
mysql> exit
```

将上一节中的 submit.html 中的如下语句。

```
<form id="form" name="form" method="post" action="addmessage.py">
```

修改为如下所示。

```
<form id="form" name="form" method="post" action="addmessage.psp">
```

编写如下所示的添加留言页面 addmessage.psp。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<title>添加成功</title>
</head>
<body>
<%
import MySQLdb                                          # 导入 MySQLdb 模块
import datetime
name = req.form['name']                                 # 获取表单内容
mail = req.form['email']
site = req.form['site']
content = req.form['content']
now = datetime.datetime.now()
time = now.strftime('%Y-%m-%d %H:%M:%S')
db = MySQLdb.connect(host='localhost',                  # 连接到数据库
       user='root',
       passwd='python',
```

```
        db='message',
        charset='gb2312')                                    # 设置字符编码
cur = db.cursor()
cur.execute("INSERT INTO message VALUES('%s','%s','%s','%s','%s')" %
(name,mail,site,content,time))
db.commit()
cur.close()
db.close()
%>
<h1>添加成功</h1>
<br>
<a href=show.psp>单击查看留言</a>
</body>
</html>
```

编写如下所示的查看留言页面 show.psp。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<title>use Python in ASP</title>
</head>
<body>
<center>
<h1>所有留言</h1>
</center>
<hr />
<%
import MySQLdb                                               # 导入 MySQLdb 模块
db = MySQLdb.connect(host='localhost',                       # 连接到数据库
        user='root',
        passwd='python',
        db='message',
        charset='gb2312')                                    # 设置字符编码
cur = db.cursor()
cur.execute('select * from message')
results = cur.fetchall()
for result in results:                                       # 输出留言内容
    req.write( '姓名: %s' % result[0].encode('gb2312') )
    req.write( '<br>' )
    req.write( '时间: %s' % result[4] )
    req.write( '<br>' )
    req.write( '邮箱: %s' % result[1].encode('gb2312') )
    req.write( '<br>' )
    req.write( '网站: %s' % result[2].encode('gb2312') )
    req.write( '<br>' )
    req.write( '留言内容:' )
    req.write( '<br>' )
    req.write( result[3].encode('gb2312') )
    req.write( '<hr />' )
cur.close()
db.close()
%>
</body>
</html>
```

将上述页面保存到 Apache 网站发布目录，如“D:\Apache Software Foundation\Apache2.2\htdocs”中，在 IE 地址栏中输入 http://127.0.0.1:82/submit.html，就可以提交、查看留言了。

17.5 本章小结

本章介绍了 Python 在 Web 方面的应用，主要介绍了 3 个方面的内容。首先介绍了 Python 提供的开源 Web 应用服务器 Zope 的安装和使用，并在此基础上介绍了 Plone 内容管理系统的安装和使用。然后介绍了在 Microsoft IIS 中使用 Python 的案例。最后介绍了在 Apache 中使用 Python 的方法。IIS 和 Apache 是目前主流的 Web 服务器，通过对本章的学习，读者应该可以在这两种 Web 服务器中使用 Python 进行简单的网站开发了。

下一章将介绍 Python 在网络编程方面的应用。

◆ ◆ ◆

第 18 章 Python 网络编程

本章包括

- 使用 socket 模块建立网络通信
- 使用 urllib、httplib 以及 ftplib
- 在局域网中传输文件
- 使用 poplib 和 smtplib 模块收发邮件

在 Python 的标准模块中，提供了对网络编程功能的支持。在 Python 中，即可以使用 socket 模块进行底层的网络编程，也可以使用 urllib、httplib、ftplib、poplib 和 smtplib 等模块针对特定的网络协议进行编程。除了 Python 的标准模块外，还可以使用 Twisted 进行网络编程。Twisted 支持多种底层协议，使用 Twisted 可以更加方便地编写网络应用程序。

18.1 使用 socket 模块

Python 中的 socket 模块提供了底层的网络接口，使用 Python 的 socket 模块可以实现网络上不同计算机之间的 socket 通信。Python 中的 socket 实现了 BSD（Berkeley Software Distribution）套接字标准。

18.1.1 网络编程概述

计算机网络实现了将分布在世界各地的不同计算机相互连接起来。计算机网络的出现使得资源共享变得简单。如今，计算机网络的飞速发展，使得以前看似天方夜谭的幻想都变成了现实。网络的出现，打破了地域的隔阂，世界一下子变得很小。现在，只需要一台能连接到 Internet 上的计算机，用鼠标一点就能了解世界各地的新闻大事。

虽然网络飞速发展，使得网络时代的生活变得新鲜、多样起来，然而网络的本质却没有改变。计算机仍然依靠 socket 进行通信，现在的网络服务也都是建立在 socket 基础之上。

socket 是网络连接端点，是网络的基础。每个 socket 都被绑定到指定的 IP 和端口上。例如，第 17 章中使用 127.0.0.1:8080 来访问 Zope 构建的网站。其中 127.0.0.1 是一个特殊的 IP 地址，它总是指向本机。而 IP 地址后的 8080 则是 Zope 服务所监听的端口。

每台处于网络中的计算机都有一个 IP 地址来标识自己的位置。IP 地址是一个 32 位长的二进制数，为方便起见，IP 地址通常由小于 255 的四组整数组成，每组数之间用“.”隔开，如“123.45.67.89”，如果将其写成二进制的形式则是“01111011001011010100001101011001”。有些 IP 地址比较特别，例如 127.0.0.1，它总是指向本机，而以“192.168.*.*”开头的 IP 地址则只能用于局域网中。

IP 地址还可以和域名绑定，例如，IP 地址 82.94.164.162 绑定到了 www.python.org 域名上。使用 82.94.164.162 同样可以访问 Python 的官方网站，显然 IP 地址不像域名那样容易记忆。

除了 IP 地址，使用 socket 时还需要指定计算机端口。计算机端口的取值范围是 0～65535，其中小于 1024 的端口都是系统所保留的端口（或者是一些网络服务所使用的端口）。例如，FTP

服务使用 21 端口，Web 服务使用 80 端口。当然，系统也会使用一些编号较大的端口，如 Windows 系统中著名的 3389 端口是远程连接端口。

IP 和端口表明了计算机在网络中的位置，类似于邮政编码和详细地址。

另外，计算机之间为了进行通信，还需要遵循特定的计算机网络协议，常见的网络协议有以下几种。

- TCP/IP 协议，即传输控制协议/互联网协议，其用于在安装了不同硬件和不同操作系统的计算机之间实现可靠的网络通信。其中，TCP 协议用于保证数据包传输的可靠性；IP 协议用于保证数据包能被传送到目标计算机。
- NetBIOS 协议，是由 IBM 公司开发的，主要用于小型局域网。NetBIOS 协议为程序提供了请求低级服务的统一的命令接口，几乎所有的局域网都是在 NetBIOS 协议基础上工作的。
- FTP 协议，即文件传输协议。FTP 是 Internet 中使用最多的文件传输协议。通过 FTP 协议可以使用客户端从 FTP 服务器中下载或上传各种文件。
- Telnet 协议，即远程登录协议，使用 Telnet 协议可以登录到远程计算机。目前仍有很多的 BBS 可以使用 Telnet 登录。
- HTTP 协议，即超文本传输协议，其用于传送 WWW 方式的数据。第 17 章中所创建的 Web 应用都是基于 HTTP 协议的。
- PPP 协议，即点对点协议，其主要用来创建电话线路以及 ISDN 拨号接入 ISP 的连接，具有多种身份验证方法、数据压缩和加密以及通知 IP 地址等功能。
- PPPoE 协议，即以太网上的点对点协议，其广泛用于 ADSL 接入方式，也就是常说的宽带。通过 PPPoE 技术和宽带调制解调器，用户可以创建虚拟拨号连接，连接到 Internet 上。

18.1.2 使用 socket 模块建立网络通信

使用 Python 中 socket 模块提供的 socket 对象的方法，可以在计算机之间建立连接。一般来说，使用 socket 创建的通信应有服务端和客户端，服务端首先建立一个 socket，并等待客户端的连接。客户端建立与服务端的 socket 连接，当连接成功后，客户端和服务端就可以使用 socket 进行通信。

1. socket 模块简介

使用 socket 模块时，应首先使用 socket()函数，创建一个 socket 对象，然后即可使用 socket 对象的方法创建连接。socket()函数的原型如下。

```
socket( family, type, proto)
```

其参数含义如下。

- family　　地址系列，可选参数。默认为 AF_INET，也可以是 AF_INET6 或 AF_UNIX。
- type　　socket 类型，可选参数。默认为 SOCK_STREAM。
- proto　　协议类型，可选参数。

创建好 socket 对象后，可以使用 socket 对象的 bind 方法绑定 IP 地址和端口。bind 方法的原型如下。

```
bind(address)
```

其参数含义如下。

- address　　由 IP 地址和端口组成的元组，如 "('127.0.0.1',1051)"。如果 IP 地址为空，则表示本机。

使用 socket 对象的 listen 方法可以监听由 socket 对象创建的连接。其函数原型如下。

```
listen(backlog)
```

其参数含义如下。

- backlog　　指定连接队列数，最小值为 1，最大值由所使用的操作系统决定，一般情况下为 5。

使用 socket 对象的 connect 和 connect_ex 都可以连接到服务端，不同的是，当连接不成功时，connect 将返回一个错误，而 connect_ex 则引发一个异常。其函数原型如下。

```
connect(address)
connect_ex(address)
```

其参数含义相同，如下所示。

- address　　由 IP 地址和端口组成的元组。

使用 socket 对象的 accept 方法可以接受来自客户端的连接，accept 方法将返回一个新的 socket 对象和客户端的地址。使用 socket 对象的 recv 和 recvfrom 方法都可以从 socket 对象获取数据，不同的是，recvfrom 方法返回所接受的字符串和地址，而 recv 方法仅返回字符串，其原型如下。

```
recv(bufsize, flags)
recvfrom(bufsize, flags)
```

其参数含义相同，如下所示。

- bufsize　　指定接受缓冲区大小。
- flags　　接受标志，可选参数。

使用 socket 对象的 send 和 sendall 方法都可以向已经连接的 socket 发送数据，不同的是，sendall 将一次发送完全部数据。其原型如下。

```
send(string, flags)
sendall(string, flags)
```

其参数含义相同，如下所示。

- string　　所发送的数据。
- flags　　发送标志，可选参数。

使用 socket 对象的 sendto 方法可以向一个未连接的 socket 发送数据，其参数原型如下。

```
sendto(string, flags, address)
```

其参数含义如下。

- string　　所发送的数据。
- flags　　发送标志，可选参数。
- address　　由 IP 地址和端口组成的元组。

使用 socket 对象的 makefile 方法可以将 socket 关联到文件对象上，其原型如下。

```
makefile(mode, bufsize)
```

其参数含义如下。

- mode　　文件模式，可选参数。
- bufsize　　缓冲区大小，可选参数。

当完成通信后，应使用 socket 对象的 close 方法关闭网络连接。

2. 建立服务端

在使用 socket 模块建立一个简单的服务端时，首先应创建一个 socket 对象，使用 socket 对象的 bind 方法绑定 IP 地址和端口，然后使用 socket 对象的 listen 方法监听 socket 连接，最后进入循环，等待客户端的连接。

下面所示的 server.py 脚本是使用 socket 模块创建一个简单的服务端程序。

```
# -*- coding:utf-8 -*-
# file: server.py
#
import tkinter
import threading
import socket
class ListenThread(threading.Thread):                    # 监听线程
    def __init__(self,edit,server):
        threading.Thread.__init__(self)
        self.edit = edit                                 # 保存窗口中的多行文本框
        self.server = server
    def run(self):                                       # 进入监听状态
        while 1:                                         # 使用 while 循环等待连接
            try:                                         # 捕获异常
                client, addr = self.server.accept()      # 等待连接
                self.edit.insert(tkinter.END,            # 向文本框中输出状态
                        '连接来自:%s:%d\n' % addr)
                data = client.recv(1024)                 # 接收数据
                self.edit.insert(tkinter.END,            # 向文本框中输出数据
                        '收到数据:%s \n' % data)
                client.send(str('I GOT: %s' % data).encode())  # 发送数据
```

```
                client.close()                                  # 关闭同客户端的连接
                self.edit.insert(tkinter.END,                   # 向文本框中输出状态
                        '关闭客户端\n')
            except:                                             # 异常处理
                self.edit.insert(tkinter.END,                   # 向文本框中输出状态
                        '关闭连接\n')
                break                                           # 结束循环
class Control(threading.Thread):                                # 控制线程
    def __init__(self, edit):
        threading.Thread.__init__(self)
        self.edit = edit                                        # 保存窗口中的多行文本框
        self.event = threading.Event()                          # 创建 Event 对象
        self.event.clear()                                      # 清除 event 标志
    def run(self):
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                                                                # 创建 socket 连接
        server.bind(('', 1051))                                 # 绑定本机 1051 端口
        server.listen(1)                                        # 开始监听
        self.edit.insert(tkinter.END,'正在等待连接\n')          # 向文本框中输出状态
        self.lt = ListenThread(self.edit,server)                # 创建监听线程对象
        self.lt.setDaemon(True)
        self.lt.start()                                         # 执行监听线程
        self.event.wait()                                       # 进入等待状态
        server.close()                                          # 关闭连接
    def stop(self):                                             # 结束控制进程
        self.event.set()                                        # 设置 event 标志
class Window:                                                   # 主窗口
    def __init__(self, root):
        self.root = root
        self.butlisten = tkinter.Button(root,                   # 创建组件
                text = '开始监听', command = self.Listen)
        self.butlisten.place(x = 20, y = 15)
        self.butclose = tkinter.Button(root,
                text = '停止监听', command = self.Close)
        self.butclose.place(x = 120, y = 15)
        self.edit = tkinter.Text(root)
        self.edit.place(y = 50)
    def Listen(self):                                           # 处理按钮事件
        self.ctrl = Control(self.edit)                          # 创建控制线程对象
        self.ctrl.setDaemon(True)
        self.ctrl.start()                                       # 执行控制线程
    def Close(self):
        self.ctrl.stop()                                        # 结束控制线程
root = tkinter.Tk()
window = Window(root)
root.mainloop()
```

在 server.py 脚本中，由于是使用 while 循环监听连接，因此为了避免图形界面下假死的状态，应将 while 循环放在一个线程里执行。但 Python 并没有提供结束线程的函数或者方法，为了能够随时终止监听，所以在脚本中创建了一个控制线程。通过控制线程执行监听线程，然后控制线程进

入等待状态。当 event 被设置后，在控制线程中将关闭 socket 连接，监听也就停止了。由于监听线程已经进入监听状态，而在控制线程中关闭 socket 连接将导致异常，所以在监听线程中使用 try 捕获异常，结束循环。

3. 建立客户端

客户端的创建相对简单，只要连接指定的 IP 和端口地址，然后向服务端发送数据即可，下面所示的 client.py 脚本创建了一个简单的客户端。

```
# -*- coding:utf-8 -*-
# file: client.py
#
import tkinter
import socket
class Window:
    def __init__(self, root):                                    # 创建组件
        label1 = tkinter.Label(root, text = 'IP')
        label2 = tkinter.Label(root, text = 'Port')
        label3 = tkinter.Label(root, text = 'Data')
        label1.place(x = 5, y = 5)
        label2.place(x = 5, y = 30)
        label3.place(x = 5, y = 55)
        self.entryIP = tkinter.Entry(root)
        self.entryIP.insert(tkinter.END, '127.0.0.1')
        self.entryPort = tkinter.Entry(root)
        self.entryPort.insert(tkinter.END, '1051')
        self.entryData = tkinter.Entry(root)
        self.entryData.insert(tkinter.END, 'Hello')
        self.Recv = tkinter.Text(root)
        self.entryIP.place(x = 40, y = 5)
        self.entryPort.place(x = 40, y = 30)
        self.entryData.place(x = 40, y = 55)
        self.Recv.place(y = 115)
        self.send = tkinter.Button(root, text = '发送数据', command = self.Send)
        self.send.place(x = 40, y = 80)
    def Send(self):                                              # 按钮事件
        try:                                                     # 异常处理
            ip = self.entryIP.get()                              # 获取 IP
            port = int(self.entryPort.get())                     # 获取端口
            data = self.entryData.get()                          # 获取发送数据
            client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                                                                 # 创建 socket 对象
            client.connect((ip,port))                            # 连接服务端
            client.send(data)                                    # 发送数据
            rdata = client.recv(1024)                            # 结束数据
            self.Recv.insert(tkinter.END, 'Server:' + rdata .decode() + '\n')
                                                                 # 输出接受的数据
            client.close()                                       # 关闭连接
        except :
            self.Recv.insert(tkinter.END, '发送错误\n')
root = tkinter.Tk()
window = Window(root)
root.mainloop()
```

运行 server.py，单击【开始监听】按钮，服务端进入监听状态，如图 18-1 所示。运行 client.py，单击【发送数据】按钮，客户端即可向服务端发送数据，如图 18-2 所示。



图 18-1　服务端



图 18-2　客户端

需要注意的是，在发送数据时只能用 byte 类型进行发送，因此发送数据时需要使用 str.encode 函数将字符串 str 编码为 byte，而接收到数据以后，又需要使用 byte.decode 函数将 byte 转换为 str 进行输出。

18.1.3　在局域网中传输文件

使用 Python 中的 socket 模块可以编写一个简单的传输文件的脚本。所谓传输文件，是指通过网络将文件从一台计算机依次发送到另一台计算机的过程。因此可以修改 18.1.2 节中的例子，将发送数据部分修改成发送文件内容即可。

下面所示的 FileServer.py 脚本修改了 server.py 脚本，用于客户端传送的文件，并将其保存在当前目录中，文件名为“receive.txt”。

```
# -*- coding:utf-8 -*-
# file: FileServer.py
#
import tkinter
import threading
import socket
import os
class ListenThread(threading.Thread):                          # 创建监听线程
    def __init__(self,edit,server):
        threading.Thread.__init__(self)
        self.edit = edit                                       # 保存窗口中的多行文本框
        self.server = server
        self.file = 'receive.txt'
    def run(self):                                             # 进入监听状态
        while 1:                                               # 使用 while 循环不停监听
            try:                                               # 捕获异常
                self.client, addr = self.server.accept()    # 等待连接
                self.edit.insert(tkinter.END,                  # 向文本框中输出状态
                        '连接来自:%s:%d\n' % addr)
                self.edit.insert(tkinter.END,                  # 向文本框中输出数据
                        '开始接收数据:')
                file = os.open(self.file, os.O_WRONLY|os.O_CREAT   # 创建文件
                        |os.O_EXCL|os.O_BINARY)
                while 1:
                    rdata = self.client.recv(1024)             # 接受数据
                    if not rdata:
```

```
                    break
                os.write(file,rdata)                    # 将数据写入文件
                self.edit.insert(tkinter.END,'......') # 向文本框中输出进度
            os.close(file)                              # 关闭文件
            self.client.close()                         # 关闭与客户端的连接
            self.edit.insert(tkinter.END,               # 向文本框中输出状态
                '\n 接收完毕,关闭客户端\n')
        except:                                         # 异常处理
            self.edit.insert(tkinter.END,               # 向文本框中输出状态
                '\n 网络错误,关闭连接\n')
            break                                       # 结束循环
class Control(threading.Thread):                        # 控制线程
    def __init__(self, edit):
        threading.Thread.__init__(self)
        self.edit = edit                                # 保存窗口中的多行文本框
        self.event = threading.Event()                  # 创建 Event 对象
        self.event.clear()                              # 清除 event 标志
    def run(self):
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                                                        # 创建 socket 连接
        server.bind(('', 1051))                         # 绑定本机 1051 端口
        server.listen(1)                                # 开始监听
        self.edit.insert(tkinter.END,'正在等待连接\n')   # 向文本框中输出状态
        self.lt = ListenThread(self.edit,server)        # 创建监听线程对象
        self.lt.setDaemon(True)
        self.lt.start()                                 # 执行监听线程
        self.event.wait()                               # 进入等待状态
        server.close()                                  # 关闭连接
    def stop(self):                                     # 结束控制进程
        self.event.set()                                # 设置 event 标志
class Window:                                           # 主窗口
    def __init__(self, root):
        self.root = root
        self.butlisten = tkinter.Button(root,           # 创建组件
                text = '开始监听', command = self.Listen)
        self.butlisten.place(x = 20, y = 15)
        self.butclose = tkinter.Button(root,
                text = '停止监听', command = self.Close)
        self.butclose.place(x = 120, y = 15)
        self.edit = tkinter.Text(root)
        self.edit.place(y = 50)
    def Listen(self):                                   # 处理按钮事件
        self.ctrl = Control(self.edit)                  # 创建控制线程对象
        self.ctrl.setDaemon(True)
        self.ctrl.start()                               # 执行控制线程
    def Close(self):
        self.ctrl.stop()                                # 结束控制线程
root = tkinter.Tk()
window = Window(root)
root.mainloop()
```

下面所示的 FileClient.py 脚本修改了 client.py 脚本，用于发送文件（脚本中没有处理字符编码，因此发送的文件路径和文件名不能包含中文字符）。

```
# -*- coding:utf-8 -*-
# file: FileClient.py
#
import tkinter
import tkinter.filedialog
import socket
import os
class Window:
    def __init__(self, root):                                    # 创建组件
        label1 = tkinter.Label(root, text = 'IP')
        label2 = tkinter.Label(root, text = 'Port')
        label3 = tkinter.Label(root, text = '文件')
        label1.place(x = 5, y = 5)
        label2.place(x = 5, y = 30)
        label3.place(x = 5, y = 55)
        self.entryIP = tkinter.Entry(root)
        self.entryIP.insert(tkinter.END, '127.0.0.1')
        self.entryPort = tkinter.Entry(root)
        self.entryPort.insert(tkinter.END, '1051')
        self.entryData = tkinter.Entry(root)
        self.entryData.insert(tkinter.END, 'Hello')
        self.entryIP.place(x = 40, y = 5)
        self.entryPort.place(x = 40, y = 30)
        self.entryData.place(x = 40, y = 55)
        self.send = tkinter.Button(root, text = '发送文件', command = self.Send)
        self.openfile = tkinter.Button(root, text = '浏览', command = self.Openfile)
        self.send.place(x = 40, y = 80)
        self.openfile.place( x = 170, y = 55)
    def Send(self):                                              # 按钮事件
        try:                                                     # 异常处理
            ip = self.entryIP.get()                              # 获取 IP
            port = int(self.entryPort.get())                     # 获取端口
            filename = self.entryData.get()                      # 获取发送数据
            tt = filename.split('/')
            name = tt[len(tt)-1]
            client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                                                                 # 创建 socket 对象
            client.connect((ip,port))                            # 连接服务端
            client.send(name.encode())                           # 发送文件名

            file = os.open(filename, os.O_RDONLY | os.O_EXCL|os.O_BINARY)
                                                                 # 打开文件
            while 1:                                             # 发送文件
                data = os.read(file,1024)
                if not data:
                    break
                client.send(data)
            os.close(file)                                       # 关闭文件
            client.close()                                       # 关闭连接
        except Exception as e :
```

```
            print('发送错误',e)
    def Openfile(self):
        r = tkinter.filedialog.askopenfilename(title = 'Python tkinter',  # 创
建打开文件对话框
            filetypes=[('All files', '*'),('Python', '*.py *.pyw')])
        if r:
            self.entryData.delete(0, tkinter.END)
            self.entryData.insert(tkinter.END, r)

root = tkinter.Tk()
window = Window(root)
root.mainloop()
```

运行 FileServer.py 脚本，打开如图 18-3 左图所示窗口，单击【开始监听】按钮，等待客户端发送文件。接着运行 FileClient.py 脚本，单击【浏览】按钮找到要发送的文本文件，如图 18-4 所示，接着单击【发送文件】按钮，即可向服务端发送文件。同时，服务端的文本框中将显示开始接收数据的状态，数据接收完毕后将自动关闭连接，服务端状态的变化如图 18-3 右图所示。



图 18-3 FileServer 接收文件



图 18-4 FileClient 发送文件

文件传输完毕后，在服务端脚本所在目录中可以看到名为“receive.txt”的文件，其内容与客户端所传文件“test.txt”中的相同（第一行是文件名“test.txt”）。

18.2 使用 urllib、httplib 和 ftplib

Python 提供的 socket 模块主要用于底层网络协议，因此编写程序较为麻烦，这一点从 18.1 节的两个例子可以看出来。其实，通常大部分情况下的网络编程都是针对应用协议进行的。例如，常用的 HTTP 协议和 FTP 协议，这时，可以使用 Python 中的 httplib 和 ftplib 等进行访问。

18.2.1 使用 Python 访问网站

网站都是基于 HTTP 协议的，使用 Python 中的 urllib 和 httplib 都可以访问网站。其中 urllib 主要用于处理 URL（Universal Resource Locators），使用 urllib 操作 URL 可以像使用和打开本地文件一样操作，简单易用。而 httplib 则实现了对 HTTP 协议的封装。

1. urllib.request 模块简介

在 Python 3 中，对 urllib 模块进行了重组分类，将原来 Python 2 中 urllib 模块中的方法进行了细分。例如，由 urllib.request 模块提供 urlopen、urlretrieve 等方法。

使用 Python 提供的的 urllib.request 模块可以对 URL 进行处理（在 Python 2.x 中直接由 urllib

模块进行处理），例如，使用 urllib.request 模块中的 urlopen 方法可以方便地打开一个 URL。urlopen 的原型如下。

```
urlopen(url, data, proxies)
```

其参数含义如下。

- ◆ url　　要进行操作的 URL 地址。
- ◆ data　　向 URL 传递的数据，可选参数。
- ◆ proxies　　使用的代理地址，可选参数。

urlopen 将返回一个类似于 file 的对象，可以像操作文件一样使用 read、readline、close 等方法对 URL 进行操作。

使用 urllib.request 模块中的 urlretrieve 方法可以将 URL 保存为本地文件。urlretrieve 方法的原型如下。

```
urlretrieve(url, filename, reporthook, data)
```

其参数含义如下。

- ◆ url　　要保存的 URL 地址。
- ◆ filename　　指定保存的文件名，可选参数。
- ◆ reporthook　　回调函数，可选参数。
- ◆ data　　发送的数据，一般用于 POST，可选参数。

2. urllib.request 模块简介

在 Python 3 中，将 URL 编码、解码的操作归类到 urllib.request 模块。

使用 urllib.parse 模块中的 urlencode 函数可以对 URL 进行编码（在 Python 2.x 中直接由 urllib 模块进行处理），其原型如下。

```
urlencode(query, doseq)
```

其参数含义如下。

- ◆ query　　要进行编码的变量和值组成的字典。
- ◆ doseq　　可选参数，若为 True，则将为元组的值分别编码成“变量=值”的形式。

使用 urllib.parse 模块中的 quote 方法和 quote_plus 方法可以替换字符串中的特殊字符，使其符合 URL 所要求使用的字符（在 Python 2.x 中直接由 urllib 模块进行处理）。这两个方法的原型如下。

```
quote(string, safe)
quote_plus(string, safe)
```

这两个方法的参数相同，其含义如下。

- ◆ string　　要进行替换的字符串。
- ◆ safe　　可选参数，指定不需要替换的字符，默认为“/”。

使用 urllib.parse 模块中的 unquote 方法和 unquote_plus 方法可以将使用 quote 方法和 quote_plus 方法替换后的字符还原（在 Python 2.x 中直接由 urllib 模块进行处理）。这两个方法的原型如下。

```
unquote(string)
unquote_plus(string)
```

其参数含义如下。

- string　　要进行还原的字符串。

3. httplib.client 模块简介

与 urllib 模块类似，在 Python 3 中，将 httplib 模块也进行了细分，主要分为 httplib.client 模块和 httplib.server 模块。

在 Python 的 httplib.client 模块中提供了 HTTPConnection 对象和 HTTPResponse 对象。当创建一个 HTTPConnection 对象后，可以使用 request 方法向服务器发送请求。request 方法的原型如下。

```
request(method, url, body, headers)
```

其参数含义如下。

- method　　发送的操作，一般为“GET”或“POST”。
- url　　进行操作的 URL。
- body　　发送的数据。
- headers　　发送的 HTTP 头。

当向服务器发送请求后，可以使用 HTTPConnection 对象的 getresponse 方法返回一个 HTTPResponse 对象。使用 HTTPConnection 对象的 close 方法可以关闭与服务器的连接。除了使用 request 方法，还可以依次使用如下所示的方法向服务器发送请求。

```
putrequest(request, selector, skip_host, skip_accept_encoding)
putheader(header, argument, ...)
endheaders()
send(data)
```

putrequest 方法的参数含义如下。

- request　　所发送的操作。
- selector　　进行操作的 URL。
- skip_host　　可选参数，若为真，则禁止自动发送“HOST:”。
- skip_accept_encoding　　可选参数，若为真，则禁止自动发送“Accept-Encoding:headers”。

putheader 方法的参数含义如下。

- header　　发送的 HTTP 头。
- argument　　发送的参数。

send 方法的参数含义如下。

- data　　发送的数据。

httplib 模块中的 HTTPResponse 对象主要用于处理服务器对所发送请求的响应。使用 HTTPResponse 对象的 read 方法可以获得服务器响应主体，使用 HTTPResponse 对象的 getheader 方法可以获得服务器响应的 HTTP 头，其原型如下。

```
getheader(name, default)
```

其参数含义如下。

- name　　指定 HTTP 头名。
- default　　可选择参数，如果指定的 name 不存在，则获取 default 指定的 HTTP 头。

HTTPResponse 对象还具有 version、status 和 reason 等属性，用于查看 HTTP 协议的版本、状态等。

4. 使用 Python 访问网站

使用 Python 的 urllib 模块可以创建一个简单的访问网站的脚本，获取指定的页面。如果使用 GUI 库中显示 HTML 的组件，则还可以制作一个简单的 Python Web 浏览器。

使用 httplib 模块也可以访问网站，但过程要比 urllib 模块复杂。httplib 模块可以用于需要用户名和密码认证的网站，而 urlib 模块则只能简单地访问、下载页面内容。

下面所示的 httpurl.py 脚本是使用 urllib.request 模块读取网页内容。

```
# -*- coding:utf-8 -*-
# file: httpurl.py
#
import tkinter
import urllib.request

class Window:
    def __init__(self, root):
        self.root = root
        self.entryUrl = tkinter.Entry(root)                    # 创建组件
        self.entryUrl.place(x = 5, y = 15)
        self.get = tkinter.Button(root,
                text = '下载页面', command = self.Get)
        self.get.place(x = 160, y = 12)
        self.edit = tkinter.Text(root)
        self.edit.place(x=5, y = 50)
    def Get(self):
        url = self.entryUrl.get()                              # 获取 URL
        page = urllib.request.urlopen(url)                     # 打开 URL
        data = page.read()                                     # 读取 URL 内容
        self.edit.insert(tkinter.END, data)                    # 将内容输出到文本框
        page.close()
root = tkinter.Tk()
window = Window(root)
root.minsize(580,380)
root.mainloop()
```

运行 httpurl.py 后，将打开窗口，在左上方的文本框中输入网址 http://www.python.org 后，单击【下载页面】按钮，下面的多行文本框中将显示 python 官方网上的 html 代码，如图 18-5 所示。



图 18-5 使用 urllib 下载页面

18.2.2 访问 FTP

Python 中的 ftplib 模块提供了用于访问 FTP 的函数。使用 ftplib 模块可以在 Python 脚本中访问 FTP，完成文件的上传、下载等操作。

1. ftplib 模块简介

使用 ftplib 模块中的 FTP 类，可以创建一个 FTP 连接对象。其原型如下。

```
FTP(host, user, passwd, acct)
```

其参数含义如下。

- host　　要连接的 FTP 服务器，可选参数。
- user　　登录 FTP 服务器所使用的用户名，可选参数。
- passwd　　登录 FTP 服务器所使用的密码，可选参数。
- acct　　可选参数，默认为空。

当创建一个 FTP 连接对象以后，可以使用 set_debuglevel 方法设置调试级别。其原型如下。

```
set_debuglevel(level)
```

其参数含义如下。

- level　　调试级别，默认的调试级别为 0。

如果在创建 FTP 连接对象时未使用 HOST 参数，则可以使用 FTP 对象的 connect 方法，其原型如下。

```
connect(host, port)
```

其参数含义如下。

- host　　要连接的 FTP 服务器。
- port　　FTP 服务器的端口，可选参数。

如果在创建 FTP 对象时未使用用户名和密码，则可以使用 FTP 对象的 login 对象使用用户名和密码登录到 FTP 服务器。其原型如下。

```
login(user, passwd, acct)
```

- user　　登录 FTP 服务器所使用的用户名。
- passwd　　登录 FTP 服务器所使用的密码。
- acct　　可选参数，默认为空。

使用 FTP 对象的 getwelcome 方法可以获得 FTP 服务器的欢迎信息。使用 FTP 对象的 abort 方法可以中断文件传输。使用 FTP 对象的 sendcmd 和 voidcmd 方法可以向 FTP 服务器发送命令，这两个方法的不同之处在于 voidcmd 方法没有返回值。这两个方法的原型如下。

```
sendcmd(command)
voidcmd(command)
```

其参数含义相同，如下所示。

- command　　向服务器发送的命令字符串。

使用 FTP 对象的 retrbinary 和 retrlines 方法可以从 FTP 服务器下载文件。不同的是，retrbinary 方法使用二进制形式传输文件，而 retrlines 方法使用 ASCII 形式传输文件。其函数原型如下。

```
retrbinary(command, callback, maxblocksize, rest)
retrlines(command, callback)
```

对于 retrbinary，其参数含义如下。

- command　　传输命令，由“RETR+文件名”组成（之间有空格）。
- callback　　传输回调函数。
- maxblocksize　　设置每次传输的最大字节数，可选参数。
- rest　　设置文件续传位置，可选参数。

对于 retrlines，其参数含义如下。

- command　　传输命令。
- callback　　传输回调函数。

使用 FTP 对象的 storbinary 和 storlines 方法可以向 FTP 服务器上传文件。这两个方法的不同之处是，storbinary 方法使用二进制形式传输文件，而 storlines 方法使用 ASCII 形式传输文件。这两个方法的原型如下。

```
storbinary(command, file, blocksize)
storlines(command, file)
```

对于 storbinary，其参数含义如下。

- command　　传输命令，由“STOR+文件名”组成（之间有空格）。
- file　　本地文件句柄。
- blocksize　　设置每次读取文件最大字节数，可选参数。

对于 storlines，其参数含义如下。

- command　　传输命令。
- file　　本地文件句柄。

使用 FTP 对象的 set_pasv 方法可以设置传输模式，其原型如下。

```
set_pasv(boolean)
```

其参数含义如下。

- boolean　　如果为 True，则为被动模式；如果为 False，则为主动模式。

使用 FTP 对象的方法 dir 方法可以获取当前目录中的内容列表。
使用 FTP 对象的 rename 方法可以修改 FTP 服务器中的文件名。其原型如下。

```
rename(fromname, toname)
```

其参数含义如下。

- fromname　　原来文件名。
- toname　　重命名后的文件名。

使用 FTP 对象的 delete 方法可以从 FTP 服务器上删除文件，其原型如下。

```
delete(filename)
```

其参数含义如下。

- filename　　要删除的文件名。

使用 FTP 对象的 cwd 方法可以改变当前目录，其原型如下。

```
cwd(pathname)
```

其参数含义如下。

- pathname　　要进入目录的路径。

使用 FTP 对象的 mkd 方法可以在 FTP 服务器上创建目录，其原型如下。

```
mkd(pathname)
```

其参数含义如下。

◆ pathname　　要创建目录的路径。

使用 FTP 对象的 pwd 方法可以获得当前目录。

使用 FTP 对象的 rmd 方法可以删除 FTP 服务器上的目录，其原型如下。

```
rmd(dirname)
```

其参数含义如下。

◆ dirname　　要删除的目录。

使用 FTP 对象的 size 方法可以获得文件的大小，其原型如下。

```
size(filename)
```

其参数含义如下。

◆ filename　　文件名。

使用 FTP 对象的 quit 和 close 方法可以关闭与 FTP 服务器的连接。

2. 使用 Python 访问 FTP

Python 的 ftplib 模块提供了完整的用于 FTP 协议的函数和方法，使用 ftplib 模块可以制作一个简单的类似于 Windows 自带的 FTP 客户端。

下面所示的 pyftp.py 脚本是使用 ftplib 模块创建了一个简单的 FTP 客户端。

```
# -*- coding:utf-8 -*-
# file: pyftp.py
#
from ftplib import FTP                                  # 从 ftplib 模块中导入 FTP
bufsize = 1024                                          # 设置缓冲区大小
def Get(filename):                                      # 下载文件
    command = 'RETR ' + filename
    ftp.retrbinary(command, open(filename,'wb').write, bufsize)
    print('下载成功')
def Put(filename):                                      # 上传文件
    command = 'STOR ' + filename
    filehandler = open(filename,'rb')
    ftp.storbinary(command,filehandler,bufsize)
    filehandler.close()
    print('上传成功')
def PWD():                                              # 获取当前目录
    print(ftp.pwd())
def Size(filename):                                     # 获取文件大小
    print(ftp.size(filename))
def Help():                                             # 输出帮助
    print('''
    ================================
        Simple Python FTP
```

```
    ================================
    cd        进入文件夹
    delete        删除文件
    dir       获取当前文件列表
    get       下载文件
    help          帮助
    mkdir         创建文件夹
    put       上传文件
    pwd       获取当前目录
    rename        重命名文件
    rmdir         删除文件夹
    size          获取文件大小
    ''')
server = input('请输入 FTP 服务器地址:')                    # 获取服务器地址
ftp = FTP(server)                                          # 连接到服务器地址
username = input('请输入用户名:')                           # 获取用户名
password = input('请输入密码:')                             # 获取字典
ftp.login(username,password)                               # 登录 FTP
print(ftp.getwelcome())                                    # 获取欢迎信息
actions  = {'dir':ftp.dir, 'pwd': PWD, 'cd':ftp.cwd, 'get':Get,# 命令与对应的函数字典
        'put':Put, 'help':Help, 'rmdir': ftp.rmd,
        'mkdir': ftp.mkd, 'delete':ftp.delete,
        'size':Size, 'rename':ftp.rename}
while True:                                                # 命令循环
    print('pyftp>', )                                      # 输出提示符
    cmds = input()                                         # 获取输入
    cmd = str.split(cmds)                                  # 将输入按空格分割
    try:                                                   # 异常处理
        if len(cmd) == 1:                                  # 判断命令是否有参数
            if str.lower(cmd[0]) == 'quit':                # 如果命令为 quit 则退出循环
                break
            else:
                actions[str.lower(cmd[0])]()               # 调用与命令对应的函数
        elif len(cmd) == 2:                                # 处理命令有一个参数的情况
            actions[str.lower(cmd[0])](cmd[1])             # 调用与命令对应的函数
        elif len(cmd) == 3:                                # 处理命令有两个参数的情况
            actions[str.lower(cmd[0])](cmd[1],cmd[2])      # 调用与命令对应的函数
        else:
            print('输入错误')
    except:
        print('命令出错')
ftp.quit()                                                 # 端口连接
```

在命令窗口运行 pyftp.py 后，将要求输入 FTP 服务器的地址、用户名和密码，都输入正确后，将完成登录，显示一个“pyftp>”提示符，等待用户输入命令，这时输入“dir”和“pwd”这两个命令后的结果如图 18-6 所示。



图 18-6 创建的 FTP 客户端



要想运行 pyftp.py 脚本，则需要有一个 FTP 服务器及登录该服务器的用户名和密码，如果没有互联网中的 FTP 服务器，也可在本地计算机中通过 IIS 配置一个 FTP 服务器进行测试。

18.3 使用 poplib 和 smtplib 模块收发邮件

Python 中的 poplib 模块和 smtplib 模块提供了对 POP3 协议和 SMTP 协议的支持，使用 POP3 协议可以登录 E-mail 服务器收取邮件，使用 SMTP 协议则可通过 E-mail 服务器发送邮件。

18.3.1 检查 E-mail

一般的邮箱服务都提供了 POP3 收取邮件的方式，Outlook 等 E-mail 客户端就是使用 POP3 协议收取邮箱中的邮件的。使用 Python 的 poplib 模块可以编写一个简单的收取邮件的客户端脚本。

1. poplib 模块简介

使用 poplib 模块中的 POP3 类可以创建一个 POP3 对象实例。其原型如下。

```
POP3(host, port)
```

其参数含义如下。

- host　　POP3 邮件服务器。
- port　　服务器端口，可选参数，默认为 110。

当创建一个 POP3 对象实例后，可以使用其 user 方法向 POP3 服务器发送用户名。其原型如下。

```
user(username)
```

其参数含义如下。

- usrname　　登录服务器的用户名。

使用 POP3 对象的 pass_方法（注意，pass 后面有一个下画线字符）可以向 POP3 服务器发送密码。其原型如下。

```
pass_(password)
```

其参数含义如下。

- password　　登录服务器的密码。

当登录服务器后，可以使用 POP3 对象的 getwelcome 方法获取服务器的欢迎信息。使用 POP3 对象的 set_debuglevel 方法可以设置调试级别。其原型如下。

```
set_debuglevel(level)
```

其参数含义如下。

- level　　调试级别。

使用 POP3 对象的 stat 方法可以获取邮箱的状态，如邮件数、邮箱大小等。使用 POP3 对象的 list 方法可以获取邮件内容列表，其原型如下。

```
list(which)
```

其参数含义如下。

- which　　可选参数，如果指定则仅列出指定的邮件内容。

使用 POP3 对象的 retr 方法可以获取指定的邮件。其原型如下。

```
retr(which)
```

其参数含义如下。

- which　　指定要获取的邮件。

使用 POP3 对象的 dele 方法可以删除指定的邮件。其原型如下。

```
dele(which)
```

其参数含义如下。

- which　　指定要删除的邮件。

使用 POP3 对象的 top 方法可以收取邮件部分内容。其原型如下。

```
top(which, howmuch)
```

- which　　指定要获取的邮件。
- howmuch　　指定获取的行数。

使用 POP3 对象的 rset 方法可以清除收件箱中邮件的删除标记。

使用 POP3 对象的 noop 方法可以保持与服务器的连接。

使用 POP3 对象的 quit 方法可以断开与服务器的连接。

2. 使用 Python 收取 E-mail

使用 Python 检查 E-mail，首先应该知道自己所使用的 E-mail 的 POP3 服务器地址和端口。对于网易 163 的邮箱，其 POP3 服务器的地址为 pop.163.com，端口默认值为 110；对于网易 126 的邮箱，其 POP3 服务器地址为 pop.126.com，端口默认值为 110。其他 E-mail 可以查看网站帮助，获取 POP3 服务器的地址和端口。

下面所示的 pypop.py 脚本可以将邮箱中的邮件下载到本地（使用时应注意，可以申请一个测试邮箱测试该脚本，避免误操作导致邮箱中邮件损失）。

```
# -*- coding:utf-8 -*-
# file: pypop.py
#
import poplib
import re
import tkinter
class Window:
    def __init__(self, root):                                    # 创建组件
        label1 = tkinter.Label(root, text = 'POP3')
        label2 = tkinter.Label(root, text = 'Port')
        label3 = tkinter.Label(root, text = '用户名')
        label4 = tkinter.Label(root, text = '密码')
        label1.place(x = 5, y = 5)
        label2.place(x = 5, y = 30)
        label3.place(x = 5, y = 55)
        label4.place(x = 5, y = 80)
        self.entryPOP = tkinter.Entry(root)
        self.entryPort = tkinter.Entry(root)
        self.entryUser = tkinter.Entry(root)
        self.entryPass = tkinter.Entry(root, show = '*')
        self.entryPort.insert(tkinter.END, '110')
        self.entryPOP.place(x = 50, y = 5)
        self.entryPort.place(x = 50, y = 30)
        self.entryUser.place(x = 50, y = 55)
        self.entryPass.place(x = 50, y = 80)
        self.get = tkinter.Button(root, text = '收取邮件', command = self.Get)
        self.get.place(x = 60, y = 120)
        self.text = tkinter.Text(root)
        self.text.place(y=150)
    def Get(self):                                               # 按钮事件
        try:                                                     # 异常处理
            host = self.entryPOP.get()                           # 获取服务器地址
            port = int(self.entryPort.get())                     # 获取端口
            user = self.entryUser.get()                          # 获取用户名
            pw = self.entryPass.get()                            # 获取密码
            pop = poplib.POP3(host)                              # 创建 POP3 实例
            pop.user(user)                                       # 登录服务器
            pop.pass_(pw)
            stat = pop.stat()                                    # 获取状态
            self.text.insert(tkinter.END,                        # 输出状态
```

```
                'Status: %d message(s), %d bytes\n' % stat)
            rx_headers  = re.compile(r"^(From|To|Subject)")# 编译正则表达式
            for n in range(stat[0]):
                response, lines, bytes = pop.top(n + 1, 10)# 收取邮件的前 10 行
                self.text.insert(tkinter.END,
                       "Message %d (%d bytes)\n" % (n+1, bytes))
                self.text.insert(tkinter.END, "-" * 30 + '\n')
                str_lines=[]
                for l in lines:                          #将接收到的 byte 转换为 str
                    str_lines.append(l.decode(encoding='gbk')) #这里使用 gbk 编码
                self.text.insert(tkinter.END,            # 输出匹配到的内容
                       "\n".join(filter(rx_headers.match, str_lines)))
                self.text.insert(tkinter.END, '\n')
                self.text.insert(tkinter.END, "-" * 30  + '\n')
        except Exception as e :
            self.text.insert(tkinter.END, '接受错误\n')
root = tkinter.Tk()
window = Window(root)
root.mainloop()
```

运行 pypop.py 脚本后，将打开如图 18-7 所示窗口，输入 POP3 服务器地址、用户名和密码之后，单击【收取邮件】按钮，脚本将从指定的 POP3 服务器获取邮件，并显示在下方的文本框中，如图 18-7 所示。



图 18-7　使用 pypop.py 收邮件

18.3.2 发送 E-mail

发送邮件一般使用的是 SMTP 协议，使用 Python 的 smtplib 模块可以使用 SMTP 协议发送邮件。要使用 SMTP 协议发送邮件，首先要登录到 SMTP 服务器。

1. smtplib 模块简介

使用 smtplib 模块的 SMTP 类可以创建一个 SMTP 对象实例。其原型如下所示。

```
SMTP(host, port, local_hostname)
```

其参数含义如下。

- host　　连接的服务器名，可选参数。
- port　　服务器端口，可选参数。
- local_hostname　　本地主机名，可选参数。

如果在创建 SMTP 对象时没有指定 host 和 port，则可以使用 SMTP 对象的 connect 方法连接到服务器。其原型如下。

```
connect(host, port)
```

其参数含义如下。

- host　　连接的服务器名，可选参数。
- port　　服务器端口，可选参数。

使用 SMTP 对象的 login 方法可以使用用户名和密码登录到 SMTP 服务器。其原型如下。

```
login(user, password)
```

其参数含义如下。

- user　　登录服务器的用户名。
- password　　登录服务器的密码。

使用 SMTP 对象的 set_debuglevel 方法可以设置调试级别。其原型如下。

```
set_debuglevel(level)
```

其参数含义如下。

- level　　调试级别。

使用 SMTP 对象的 docmd 方法可以向 SMTP 服务器发送命令。其原型如下。

```
docmd(cmd,, argstring)
```

其参数含义如下。

- cmd　　向 SMTP 服务器发送的命令。
- argstring　　命令参数，可选参数。

使用 SMTP 对象的 sendmail 方法可以发送邮件。其原型如下。

```
sendmail(from_addr, to_addrs, msg, mail_options, rcpt_options)
```

其参数含义如下。

- from_addr　　发送者邮件地址。
- to_addrs　　邮件发送地址。

- msg　　邮件内容。
- mail_options　　可选参数，邮件 ESMTP 操作。
- rcpt_options　　可选参数，RCPT 操作。

使用 SMTP 对象的 quit 方法可以断开与服务器的连接。

2. 使用 Python 发送邮件

与使用 Python 接收 E-mail 一样，使用 Python 发送 E-mail 时，也应该找到所使用的 E-mail 的 SMTP 服务器的地址和端口。对于网易 163 邮箱，其 SMTP 服务器的地址为 smtp.163.com，端口默认值为 25。对于网易 126 邮箱，其 SMTP 服务器的地址为 smtp.126.com，端口默认值为 25。

下面所示的 pysmtp.py 脚本是使用 Python 的 smtplib 模块发送邮件（为了简便起见，没有对字符进行编码，所以不能发送中文字符）。

```
# -*- coding:utf-8 -*-
# file: pysmtp.py
#
import smtplib
import tkinter
class Window:
    def __init__(self, root):                                    # 创建组件
        label1 = tkinter.Label(root, text = 'SMTP')
        label2 = tkinter.Label(root, text = 'Port')
        label3 = tkinter.Label(root, text = '用户名')
        label4 = tkinter.Label(root, text = '密码')
        label5 = tkinter.Label(root, text = '收件人')
        label6 = tkinter.Label(root, text = '主题')
        label7 = tkinter.Label(root, text = '发件人')
        label1.place(x = 5, y = 5)
        label2.place(x = 5, y = 30)
        label3.place(x = 5, y = 55)
        label4.place(x = 5, y = 80)
        label5.place(x = 5, y = 105)
        label6.place(x = 5, y = 130)
        label7.place(x = 5, y = 155)
        self.entryPOP = tkinter.Entry(root)
        self.entryPort = tkinter.Entry(root)
        self.entryUser = tkinter.Entry(root)
        self.entryPass = tkinter.Entry(root, show = '*')
        self.entryTo = tkinter.Entry(root)
        self.entrySub = tkinter.Entry(root)
        self.entryFrom = tkinter.Entry(root)
        self.entryPort.insert(tkinter.END, '25')
        self.entryPOP.place(x = 50, y = 5)
        self.entryPort.place(x = 50, y = 30)
        self.entryUser.place(x = 50, y = 55)
        self.entryPass.place(x = 50, y = 80)
        self.entryTo.place(x = 50, y = 105)
        self.entrySub.place(x = 50, y = 130)
        self.entryFrom.place(x = 50, y = 155)
        self.get = tkinter.Button(root, text = '发送邮件', command = self.Get)
        self.get.place(x = 60, y = 180)
        self.text = tkinter.Text(root)
```

```
        self.text.place(y=220)
    def Get(self):                                          # 按钮事件
        try:                                                # 异常处理
            host = self.entryPOP.get()                      # 获取服务器地址
            port = int(self.entryPort.get())                # 获取端口
            user = self.entryUser.get()                     # 获取用户名
            pw = self.entryPass.get()                       # 获取密码
            fromaddr = self.entryFrom.get()                 # 获取发件人
            toaddr = self.entryTo.get()                     # 获取收件人
            subject = self.entrySub.get()                   # 获取主题
            text = self.text.get(1.0, tkinter.END)          # 获取邮件内容
            msg = ("From: %s\nTo: %s\nSubject: %s\n\n" % (fromaddr, toaddr, subject)) # 生成邮件头
            msg = msg + text
            smtp = smtplib.SMTP(host,port)                  # 连接服务器
            smtp.set_debuglevel(1)                          # 设置调试级别
            smtp.login(user,pw)                             # 登录服务器
            smtp.sendmail(fromaddr, toaddr, msg)            # 发送邮件
            smtp.quit()                                     # 断开服务器
        except Exception as e:
            self.text.insert(tkinter.END, '发送错误\n')
root = tkinter.Tk()
window = Window(root)
root.minsize(600,480)
root.mainloop()
```

执行 pysmtp.py 脚本后，将出现如图 18-8 所示窗口，输入 SMTP 服务器、用户名、密码、收件人、主题、发件人和邮件内容等信息之后，单击【发送邮件】按钮，即可将下方文本框中输入的内容作为邮件发送给收件人。



图 18-8　使用 pysmtp.py 发送邮件

18.4 本章小结

本章介绍了 Python 在网络编程方面的应用，首先介绍了 socket 模块的使用，并通过 socket 模块编写了一个局域网文件传送的案例。接着介绍了通过 urllib、httplib、ftplib 模块访问网站、FTP 服务器的操作，最后还介绍了通过 poplib 和 smtplib 模块编写收发邮件脚本的方法。

通过本章介绍的这些内置模块，读者可以编写出常用的网络应用程序。不过，本章并没有介绍如何处理从网络中获取的数据，这是下一章的内容，下一章将介绍处理 HTML 和 XML。

◆ ◆ ◆

第 19 章　处理 HTML 与 XML

本章包括

- 用 Python 获取页面图片地址
- XML 基础
- 用 Python 编写简单的 RSS 阅读器
- 用 Python 查看天气预报
- 使用 Python 处理 XML

超文本标记语言 HTML 主要用于 Web，虽然在服务器端可以使用多种技术，但在客户端，多数情况下都是使用 HTML。可扩展标记语言 XML 的用途也很为广泛，XML 可以用于创建文件格式，如 PyGTK 中使用的资源文件。在 Python 中可以使用标准的模块对 HTML 和 XML 进行处理。

19.1　处理 HTML

在 Python 中，可以使用 html 模块处理 HTML，获取网页中感兴趣的内容。html 模块中的子模块 html.parser 提供了对 HTML 标记处理的方法。例如，可以使用 html.parser 模块中的 HTMLParser 类对 HTML 进行解析。如果有些内容不能使用 HTMLParser 处理，则还可以自己编写正则表达式进行匹配。

在 Python 3.2 之前，HTMLParser 类位于 HTMLParser 模块中，在 Python 3.2 之后，对处理 HTML 的模块进行了细化，因此就有了 html.parser 模块，而 HTMLParser 类被收归到 html.parser 模块中。

19.1.1　HTMLParser 类简介

在使用 html.parser 模块处理 HTML 时，首先应继承 html.parser 模块中的 HTMLParser 类，然后重载相关的处理方法。使用 HTMLParser 对象的 feed 方法可以向 HTMLParser 传递数据。其原型如下。

```
feed(data)
```

其参数含义如下。

- data　　传递的数据。

当向 HTMLParser 对象传递数据后，就开始对数据进行处理。使用 HTMLParser 对象的 close 方法可以强制处理 feed 方法存在缓冲区的数据。使用 HTMLParser 对象的 reset 方法可以重新设置对象实例，进行新一轮的数据处理。使用 HTMLParser 对象的 getpos 方法可以获得当前处理的行号和偏移位置。

在使用 HTMLParser 处理 HTML 过程中，遇到某些标记或者数据就会调用相应的方法。一般情况下，在脚本中需要重载这些方法，以完成对 HTML 的处理。当 HTMLParser 每遇到一个起始标记

时，会调用handle_starttag方法。其原型如下。

```
handle_starttag(tag, attrs)
```

其参数含义如下。

- tag　　HTMLParser遇到的标记。
- attrs　　标记的属性。

当HTMLParser遇到类似于
的标记时（即在标记中包含开始和结束），将调用handle_startendtag方法。其原型如下。

```
handle_startendtag(tag, attrs)
```

其参数含义如下。

- tag　　HTMLParser遇到的标记。
- attrs　　标记的属性。

当HTMLParser遇到结束标记时会调用handle_endtag方法。其原型如下。

```
handle_endtag(tag)
```

其参数含义如下。

- tag　　HTMLParser遇到的结束标记。

使用HTMLParser的handle_data方法可以处理标记间的数据。其原型如下。

```
handle_data(data)
```

其参数含义如下。

- data　　标记间的数据。

当HTMLParser遇到HTML中的注释时将调用handle_comment方法。其原型如下。

```
handle_comment(data)
```

其参数含义如下。

- data　　注释内容。

19.1.2 获取页面图片地址

根据前面的介绍可以知道，在Python的html.parser模块中提供了处理标记的方法，使用这些方法就可以处理网页中的HTML标记了。例如，由于在网页中，图片都是以“”标记嵌入的，因此要获取页面中图片的地址，只需要处理网页中的“”标记即可。

下面所示的GetImage.py脚本重载了HTMLParser类的handle_starttag方法，对“”

标记进行处理后，将分别获得 GIF 和 JPG 图片的地址。

需要注意的是，以下脚本由于没有对地址做进一步的处理，因此脚本获得的可能是图片的相对地址。

```
# -*- coding:utf-8 -*-
# file: GetImage.py
#
import tkinter
import urllib.request
from  html.parser import HTMLParser
class MyHTMLParser(HTMLParser):                    # 创建 HTML 解析类
    def __init__(self):
        HTMLParser.__init__(self)
        self.gifs = []                             # 创建列表，保存 gif
        self.jpgs = []                             # 创建列表，保存 jpg
    def handle_starttag(self, tags, attrs):        # 处理起始标记
        if tags == 'img':                          # 处理图片
            for attr in attrs:
                for t in attr:
                    if 'gif' in t:
                        self.gifs.append(t)        # 添加到 gif 列表
                    elif 'jpg' in t:
                        self.jpgs.append(t)        # 添加到 jpg 列表
                    else:
                        pass
    def get_gifs(self):                            # 返回 gif 列表
        return self.gifs
    def get_jpgs(self):                            # 返回 jpg 列表
        return self.jpgs
class Window:
    def __init__(self, root):
        self.root = root                           # 创建组件
        self.label = tkinter.Label(root, text = '输入 URL:')
        self.label.place(x = 5, y = 15)
        self.entryUrl = tkinter.Entry(root,width = 30)
        self.entryUrl.place(x = 65, y = 15)
        self.get = tkinter.Button(root,
                text = '获取图片', command = self.Get)
        self.get.place(x = 280, y = 15)
        self.edit = tkinter.Text(root,width = 470,height = 600)
        self.edit.place(y = 50)
    def Get(self):
        url = self.entryUrl.get()                  # 获取 URL
        page = urllib.request.urlopen(url)         # 打开 URL
        data = page.read()                         # 读取 URL 内容
        parser = MyHTMLParser()                    # 生成实例对象
        parser.feed(data.decode())                 # 处理 HTML 数据
        self.edit.insert(tkinter.END, '====GIF====\n')      # 输出数据
        gifs = parser.get_gifs()
        for gif in gifs:
            self.edit.insert(tkinter.END, gif + '\n')
        self.edit.insert(tkinter.END, '===========\n')
        self.edit.insert(tkinter.END, '====JPG====\n')
```

```
        jpgs = parser.get_jpgs()
        for jpg in jpgs:
            self.edit.insert(tkinter.END, jpg + '\n')
        self.edit.insert(tkinter.END, '==========\n')
        page.close()
root = tkinter.Tk()
window = Window(root)
root.minsize(600,480)
root.mainloop()
root.mainloop()
```

运行 GetImage.py 脚本后，在文本框中输入网址 http://www.python.org，单击【获取图片】按钮，将对网址进行处理，获取网页中图片的地址，如图 19-1 所示。由于很多网站使用的 HTML 并不是标准的语法格式，因此，使用 HTMLParser 处理时，不一定能够获得所有的图片。



图 19-1 获取网站图片地址

19.1.3 查看天气预报

现在很多网站都提供了免费的天气预报服务，如果每次都要登录这些网站才能查看到天气预报，则显得有些麻烦。使用 Python 编写一个脚本，可以自动获取这些网站提供的免费天气预报的内容。这样就不用打开相应的网站，只需要在本地计算机中执行一下 Python 脚本就可查看到天气预报了。

要查询天气预报，当然是以中央气象台的数据最为权威，中央气象台通过“中国天气”网站 http://www.weather.com.cn/对外发布全国各地天气预报，国内各大门户网站的天气预报数据都是从这个网站获取的。打开“中国天气”网站，查看北京的天气，可看到如图 19-2 所示的网页。

从图 19-2 中可以看到，在这个网站中有很完整的天气预报数据，并且在 IE 浏览器的地址栏中还会发现，北京的天气预报网址为：http://www.weather.com.cn/weather/101010100.shtml，即最后出现的是一串数字。再查询上海的天气预报，可以发现其网址为：http://www.weather.com.cn/weather/101020100.shtml。也就是说，每个城市对应着一个数字编码，如果要查询国内各城市的天气预报，则必须知道城市对应的编码。



图 19-2　北京的天气预报

对于城市编码这个数据，可以从网站上收集到，将其保存到一个文本文件中，查询时从文件中读取即可。例如，将收集到的城市编码按以下格式保存到 city.txt 文件中。

```
北京,101010100|北京海淀,101010200|北京朝阳,101010300|北京顺义,101010400|北京怀
柔,101010500|北京通州,101010600|北京昌平,101010700|北京延庆,101010800|北京丰
台,101010900|北京石景山,101011000|北京大兴,101011100|北京房山,101011200|北京密
云,101011300|北京门头沟,101011400|北京平谷,101011500|上海,101020100|上海闵
行,101020200|上海宝山,101020300|上海嘉定,101020500|…… ……
```

在上面的数据格式中，每一个区域名称和编码之间都用逗号分隔，而区域之间用竖线分隔。这样做的好处是可用 Python 中的 split 函数来分割数据，具体方法详见后面的代码。

知道城市编码后，就可以通过城市编码去访问对应的网页，得到该城市的天气预报数据。首先想到的方法当然是用 urllib.request 模块的 urlopen 方法打开对应的网页，获取 HTML 数据，然后进行分析。不过，这里对 HTML 进行分析的过程非常麻烦。其实，“中国天气”网还提供了另外一种方法：可以向访问者返回 JSON 数据，这些数据是有结构的，可以方便地解析。

例如，通过以下网址可以得到指定城市的实时天气：

http://www.weather.com.cn/data/sk/城市编码.html

北京编码为 101010100，可在 IE 浏览器中输入网址 http://www.weather.com.cn/data/sk/101010100.html 获取实时天气数据，如图 19-3 所示。



图 19-3　实时天气数据

从图 19-3 中可以看出，通过这种方法获取的数据是由大括号包围的 JSON 数据，而 Python 可以直接解析 Python 数据。

另外，还可以通过以下网址获取全天天气预报，以及今后六天的天气预报。

全天：http://www.weather.com.cn/data/cityinfo/101010100.html

六天：http://m.weather.com.cn/data/101010100.html

有了以上基础，下面就可以开始编写脚本了，具体代码如下。

```
#coding=utf-8
# file: GetWeather.py
#
import tkinter
import json
import urllib.request

#返回 dict 类型: twitter = {'image': imgPath, 'message': content}
def getCityWeather_RealTime(cityID):
    url = "http://www.weather.com.cn/data/sk/" + str(cityID) + ".html"
    try:
        stdout = urllib.request.urlopen(url)
        weatherInfomation = stdout.read().decode('utf-8')

        jsonDatas = json.loads(weatherInfomation)

        city       = jsonDatas["weatherinfo"]["city"]
        temp       = jsonDatas["weatherinfo"]["temp"]
        fx         = jsonDatas["weatherinfo"]["WD"]          #风向
        fl         = jsonDatas["weatherinfo"]["WS"]          #风力
        sd         = jsonDatas["weatherinfo"]["SD"]          #相对湿度
        tm         = jsonDatas["weatherinfo"]["time"]

        content = "#" + city + "#" + " " + temp + "℃ " + fx + fl + " " + "相对湿
度" + sd + " "  + "发布时间:" + tm+"\n"
        twitter = {'image': "", 'message': content}

    except (SyntaxError) as err:
        print("SyntaxError: " + err.args)
    except:
        print("OtherError: ")
    else:
        return twitter
    finally:
        None

#返回 dict 类型: twitter = {'image': imgPath, 'message': content}
def getCityWeatherDetail_SixDay(cityID):
    url = "http://m.weather.com.cn/data/" + str(cityID) + ".html"
    try:
        stdout = urllib.request.urlopen(url)
        weatherInfomation = stdout.read().decode('utf-8')
        jsonDatas = json.loads(weatherInfomation)

        city       = jsonDatas["weatherinfo"]["city"]
        tempF1     = jsonDatas["weatherinfo"]["tempF1"]
        weather    = jsonDatas["weatherinfo"]["img_title1"]
```

```
        img         = jsonDatas["weatherinfo"]["img1"]
        fx          = jsonDatas["weatherinfo"]["fx1"]                #风向
        cy           = jsonDatas["weatherinfo"]["index"]             #穿衣指数
        zw           = jsonDatas["weatherinfo"]["index_uv"]          #最弱    #紫外线指数
        xc           = jsonDatas["weatherinfo"]["index_xc"]          #不宜    #洗车
        tr           = jsonDatas["weatherinfo"]["index_tr"]          #很适宜  #旅游
        co           = jsonDatas["weatherinfo"]["index_co"]          #舒适    #舒适度
        cl           = jsonDatas["weatherinfo"]["index_cl"]          #较适宜  #晨练指数
        ls           = jsonDatas["weatherinfo"]["index_ls"]          #不太适宜#晾晒指数
        ag           = jsonDatas["weatherinfo"]["index_ag"]          #不易发  #过敏
        temp1       = jsonDatas["weatherinfo"]["temp1"]
        temp2       = jsonDatas["weatherinfo"]["temp2"]
        temp3       = jsonDatas["weatherinfo"]["temp3"]
        temp4       = jsonDatas["weatherinfo"]["temp4"]
        temp5       = jsonDatas["weatherinfo"]["temp5"]
        temp6       = jsonDatas["weatherinfo"]["temp6"]
        weather1     = jsonDatas["weatherinfo"]["weather1"]
        weather2     = jsonDatas["weatherinfo"]["weather2"]
        weather3     = jsonDatas["weatherinfo"]["weather3"]
        weather4     = jsonDatas["weatherinfo"]["weather4"]
        weather5     = jsonDatas["weatherinfo"]["weather5"]
        weather6     = jsonDatas["weatherinfo"]["weather6"]

        if int(img) < 10:
            imgPath = "icon\d" + "0" + str(img) + ".gif"
        else:
            imgPath = "icon\d"       + str(img) + ".gif"

        content = "#" + city + "#" + "\n<指数> " + "穿衣:" + cy + "\n 紫外线:" + zw
+ "\n 洗车:" + xc \
                + "\n 旅游:" + tr + "\n 舒适度:" + co + "\n 晨练:" + cl + "\n 晾晒:" +
ls + "\n 过敏:" + ag + "\n" \
                + "\n<天气>" + "\n 1 天:" + temp1 + " " + weather1 + "\n 2 天:" + temp2
+ " " + weather2
                + "\n 3 天:" + temp3 + " " + weather3\ + "\n 4 天:" + temp4 + " " + weather4
+ "\n 5 天:"
                + temp5 + " " + weather5  + "\n 6 天:" + temp6 + " " + weather6

        twitter = {'image': imgPath, 'message': content}

    except (SyntaxError) as err:
        print("SyntaxError: " + err.args)
    except:
        print(" : ")
    else:
        return twitter
    finally:
        None

#返回 dict 类型: twitter = {'image': imgPath, 'message': content}
def getCityWeather_AllDay(cityID):
    url = "http://www.weather.com.cn/data/cityinfo/" + str(cityID) + ".html"
    try:
        stdout = urllib.request.urlopen(url)
```

```
        weatherInfomation = stdout.read().decode('utf-8')
        jsonDatas = json.loads(weatherInfomation)

        city       = jsonDatas["weatherinfo"]["city"]
        temp1      = jsonDatas["weatherinfo"]["temp1"]
        temp2      = jsonDatas["weatherinfo"]["temp2"]
        weather    = jsonDatas["weatherinfo"]["weather"]
        img1       = jsonDatas["weatherinfo"]["img1"]
        img2       = jsonDatas["weatherinfo"]["img2"]
        ptime      = jsonDatas["weatherinfo"]["ptime"]

        content = city + "," + weather + ",最高气温:" + temp1 + ",最低气温:"  + temp2
             + ",发布时间:" + ptime +'\n\n'
        twitter = {'image': "icon\d" + img1, 'message': content}

    except (SyntaxError) as err:
        print("SyntaxError: " + err.args)
    except:
        print("OtherError: ")
    else:
        return twitter
    finally:
        None

class Window:
    def __init__(self, root):
        self.citys=self.getCitys('city.txt')
        self.root = root                                          # 创建组件
        self.label = tkinter.Label(root, text = '输入城市:')
        self.label.place(x = 5, y = 15)
        self.entryCity = tkinter.Entry(root)
        self.entryCity.place(x = 65, y = 15)
        self.get = tkinter.Button(root,
                text = '获取天气', command = self.Get)
        self.get.place(x = 230, y = 15)
        self.edit = tkinter.Text(root,width = 300,height = 350)
        self.edit.place(y = 50)

    #从指定文件中返回城市与编码对照字典
    def getCitys(self,file):
        file=open(file,encoding='utf-8')
        city={}
        for c in file.read().split('|'):
            cn,cc = c.split(',')
            city.update({cn:cc})
        return city

    def Get(self):
        city = self.entryCity.get().encode('utf-8')               # 获取城市
        for k in iter(self.citys.keys()):
            if k.endswith(city.decode()) :
                CityCode=self.citys[k]
                break

        title_small = "【实时天气】\n"
        twitter = getCityWeather_RealTime(CityCode)
        self.edit.insert(tkinter.END,                             # 输出日期和城市
```

```
                title_small + twitter['message']+'\n')

        title_small = "【今日天气】\n"
        twitter = getCityWeather_AllDay(CityCode)
        self.edit.insert(tkinter.END,                                    # 输出日期和城市
                title_small + twitter['message']+'\n')

        title_small = "【今后六天】\n"
        twitter = getCityWeatherDetail_SixDay(CityCode)
        self.edit.insert(tkinter.END,                                    # 输出日期和城市
                title_small + twitter['message']+'\n')

if __name__ == '__main__':
    root = tkinter.Tk()
    window = Window(root)
    root.minsize(400,445)
    root.mainloop()
```

运行 GetWeather.py 脚本后，在文本框中输入城市名，单击【获取天气】按钮，即可在多行文本框中显示天气信息，如图 19-4 所示。



图 19-4　显示天气信息

19.2 处理 XML

HTML 不具备扩展性，而且标准也不统一，因此在解析 HTML 标签时，经常需要使用正在表达式进行处理。与 HTML 相比，XML 具有更规范的结构和语法，XML 文件更加清晰。处理 XML 则相对也要容易得多，而且 XML 具有很强的扩展性。

XML 作为具有很强扩展性的标记语言，其应用十分广泛。与 HTML 具有相对固定的标记不同，XML 基本上没有自己的标记，使用 XML 时，需要根据需求自己建立标记，这不仅不会影响 XML 的结构化，反而为数据的表示提供了方便。另外，还需注意的是，通常 HTML 标签是不区分字符大小写的，但 XML 要区分大小写。

19.2.1 XML 基础

一般来说，XML 文档可能包含以下几部分内容。

1. XML 声明

XML 声明中包含三部分内容：version、encoding 和 standalone。其中 version 用于指定所使用的 XML 规范的版本号，encoding 用于指定 XML 文档所使用的编码格式，standalone 用于指定文档是否独立。下面所示的为几种不同的 XML 声明。

```
<?xml version="1.0"?>
<?xml version="1.0" encoding="utf-8"?>
<?xml version='1.0' encoding='utf-8' standalone='no'?>
```

2. 根元素

XML 文档都具有一个根元素。根元素是 XML 文档中的第一个元素，在 XML 文档的根元素中包含了其他的元素，也就是说，XML 文档中其他元素都必须包含在根元素内。

3. 元素和属性

元素是处于“< >”中的标记，而元素属性是标记后的附加信息。在下面所示的 XML 中，image 为元素，width 和 height 为属性，“=”后以单引号包围的为属性值。属性值既可以使用单引号包围，也可以使用双引号包围。

```
<image width='640' height='480'></image>
```

需要注意的是，XML 中的标记是成对出现的，如果只有一个标记则可以写成“<one/>”的形式。

4. 字符数据

字符数据是处于元素标记间的数据。XML 文档中的主要内容是字符数据。在下面所示的 XML 中，其字符数据为“Alien”（包含在元素标记 name 中）。

```
<name>
Alien
</name>
```

5. CDATA 块

CDATA 块是使用“<![CDATA[”和“]]>”包围的内容。CDATA 块适用于大段的文本，CDATA 块中可以包含特殊的字符。如果不使用 CDATA 块的话，则特殊字符，如“<”、“>”等，应该写成实体引用的形式“&lt”、“&gt”。

6. 注释

注释是以“<!--”和“-->”包围的区间，在注释内容中不允许出现“--”。

7. 处理指令

处理指令是以“<?”和“?>”包围的数据。使用处理指令可以向处理器传递指令，使其按特定的方式处理 XML 文档中的数据。下面所示的 XML 文档是 wxPython 的资源文件。

```
<?xml version="1.0" encoding="utf-8"?>
<resource>
```

```
  <object class="wxFrame" name="frame">
    <title>wxPython XRC</title>
    <object class="wxPanel" name="panel">
      <object class="wxStaticText" name="label">
        <label>Input</label>
        <pos>50,70</pos>
      </object>
      <object class="wxTextCtrl" name="text">
        <pos>120,70</pos>
      </object>
      <object class="wxButton" name="button">
        <label>Ok</label>
        <pos>100,120</pos>
      </object>
    </object>
    <size>300,200</size>
  </object>
</resource>
```

这个 XML 文档的声明部分如下。

```
<?xml version="1.0" encoding="utf-8"?>
```

其根元素为“resource”，其中，组件都处于“object”标记之间。“object”标记的“class”属性和“name”属性指明了组件的类和组件名。组件的位置信息是通过“pos”标记间的字符数据表示的。上述 XML 文档没有包含注释 CDATA 块。由此可以看出，使用 XML 文档具有很清晰的结构，通过标记可以很容易地明白其所表示的含义。

19.2.2 文档类型定义

DTD（Document Type Definition），即文档类型定义，主要用于描述 XML 文档包含的内容和 XML 文档的布局结构。使用 DTD 可以验证 XML 文档结构的正确性。

在 XML 文档中，可以在 XML 文档头使用 URL 地址指定 XML 文档所使用的 DTD，具体如下。

```
<!DOCTYPE example SYSTEM "http://www.w3c.com/dtd/portal.dtd">
```

除此之外，DTD 还可以直接包含在 XML 文档中。

通过阅读 DTD 即可知道 XML 文档中元素的安排，以及元素所具有的属性等。

DTD 的语法很简单，在 DTD 中包含了元素的声明和元素的属性定义。其中元素的声明形式如下。

```
<!ELEMENT 元素名 元素内容模型>
```

元素内容模型可以使用以下几种形式。

1. #PCDATA

元素可能包含字符数据和 XML 元素对中的普通文本。其使用形式如下。

```
<!ELEMENT element-name (#PCDATA)>
```

2. EMPTY

空元素，不包含内容，写成<tag/>的形式。其使用形式如下。

```
<!ELEMENT image EMPTY>
```

ANY　　可以包含任何元素或者字符数据。其使用形式如下。

```
<!ELEMENT image ANY>
```

3. 子元素

元素所包含的子元素，可以使用类似于正则表达式中的元字符来表示子元素的数目或者顺序等。可以使用的元字符如表19-1所示。

表19-1　XML中的元字符

符号	描述
*	零个或多个元素
+	一个或多个元素
?	零个或一个元素
,	元素按给定的顺序出现
\|	包含的元素可替换
()	组合元素

下面所示的元素定义表示，元素包含零个或一个A元素。

```
<!ELEMENT NAME A+>
```

下面所示的元素定义表示，元素包含A、B两个元素，其中，A元素处于B元素之前。

```
<!ELEMENT NAME (A,B)>
```

一般来说，元素的属性定义应紧跟元素声明之后，但不是必须的。也可将元素属性定义和元素定义分成两部分。元素属性定义形式如下。

```
<!ATTLIST 元素名
属性名1 属性数据类型 属性行为
属性名2 属性数据类型 属性行为
属性名3 属性数据类型 属性行为
.....
>
```

属性数据类型可以是以下几种。

- CDATA　　任意的文本字符串，可以包含空格等。
- NMTOKEN　　第一个字母必须是字母、任意字符或下画线的字符串。
- NMTOKENS　　和NMTOKEN类似，但是字符中应包含空格。
- ID　　表示该值在XML文档中唯一。
- IDREF　　指向XML文档中另一元素的ID。
- IDREFS　　和NMTOKENS类似，包含用空格分割的多个ID引用。
- ENTITY　　实体名。
- 枚举类型　　列举出属性所有可能的值，不同的值之间用“|”分割。

属性的行为类型如表 19-2 所示。

表 19-2 属性的行为类型

属性行为	描述
"default"	默认值
#IMPLIED	属性为可选的
#REQUIRED	属性为必需的
#FIXED	属性值为固定的

19.2.3 命名空间

XML 的命名空间是为了避免元素名和属性名的冲突。使用 XML 的命名空间可以指定 XML 元素名和属性名的使用范围，这样就可以在一定程度上避免名字冲突。

使用命名空间的元素名或属性名都是由两部分组成的，第一部分是命名空间前缀，第二部分是元素名或者属性名。这两部分由 ":" 分割。

命名空间的声明可以使用直接定义和默认定义两种方式。其中，直接定义命名空间的形式如下。

```
xmlns：命名空间前缀=命名空间名
```

默认定义命名空间的形式如下。

```
xmlns=命名空间名
```

使用默认的命名空间定义可以省略命名空间前缀部分。下面所示的 XML 使用了命名空间。

```
<ctgu:menmber xmlns:ctgu = "http://www.example.org/example.dta">
    <ctgu:item>
        <ctgu:name>Tom</ctgu:name>
        <ctgu:ID>2002105101</ctgu:ID>
        <ctgu:Phone>6390001</ctgu:Phone>
        <ctgu:Zip>443002</ctgu:Zip>
    </ctgu:item>
    <ctgu:item>
        <ctgu:name>Jack</ctgu:name>
        <ctgu:ID>2002105102</ctgu:ID>
        <ctgu:Phone>6390002</ctgu:Phone>
        <ctgu:Zip>443002</ctgu:Zip>
    </ctgu:item>
    <ctgu:item>
        <ctgu:name>Kate</ctgu:name>
        <ctgu:ID>2002105103</ctgu:ID>
        <ctgu:Phone>6390003</ctgu:Phone>
        <ctgu:Zip>443002</ctgu:Zip>
    </ctgu:item>
</ctgu:menmber>
```

19.3 使用 Python 处理 XML

在 Python 中，提供了许多标准模块用于处理 XML 文档。例如，使用 Expat 分析器的

xml.parsers.expat 模块、使用 SAX 分析器的 xml.sax 模块和使用 DOM 的 xml.dom 模块。其中，xml.parsers.expat 和 xml.sax 模块与 html.parser 模块中的 HTMLParser 类似，都是基于事件的方式对 XML 文档进行分析。

19.3.1 使用 xml.parsers.expat 处理 XML

在 Python 中，使用 xml.parsers.expat 处理 XML 时，应首先使用 ParserCreate 创建一个 XMLParser 实例对象。其原型如下。

```
ParserCreate(encoding, namespace_separator)
```

其参数含义如下。

- encoding　　XML 文档的编码，可选参数。
- namespace_separator　　XML 文档的命名空间，可选参数。

当创建 XMLParser 对象后，需要使用 Parse 或者 ParseFile 方法向其传递要处理的 XML 数据或 XML 文档。其原型如下。

```
Parse(data, isfinal)
ParseFile(file)
```

对于 Parse，其参数含义如下。

- data　　要进行处理的 XML 数据。
- isfinal　　当最后一次调用该方法时，isfinal 应为 True，可选参数。

对于 ParseFile，其参数含义如下。

- file　　打开的文件对象。

在使用 XMLParser 处理 XML 的过程中，当遇到相应的事件时，XMLParser 会调用相应的事件处理方法。在使用 XMLParser 时，可以通过继承创建新类，重载需要的处理方法，也可以直接使用 XMLParser，自己编写函数，将其赋值给 XMLParser 的处理方法。当 XMLParser 遇到 XML 文档声明时，会调用 XMLParser 方法，其原型如下。

```
XmlDeclHandler(version, encoding, standalone)
```

其参数含义如下。

- version　　XML 规范的版本。
- encoding　　XML 文档的编码。
- standalone　　XML 文档的 standalone 属性值。

当 XMLParser 遇到文档类型定义开始时，将调用 StartDoctypeDeclHandler 方法；当文档结束时，将调用 EndDoctypeDeclHandler 方法，这两个方法的原型如下。

```
StartDoctypeDeclHandler(doctypeName, systemId, publicId, has_internal_subset)
EndDoctypeDeclHandler()
```

对于 StartDoctypeDeclHandler 方法，其参数含义如下。

- doctypeName　　DTD 名称。
- systemId　　系统标识。
- publicId　　公共标识。
- has_internal_subset　　如果 XML 文档中包含 DTD，则 has_internal_subset 为真。

当 XMLParser 遇到 DTD 中元素声明时，将调用 ElementDeclHandler 方法。其原型如下。

```
ElementDeclHandler(name, model)
```

其参数含义如下。

- name　　元素名。
- model　　元素内容模型。

当 XMLParser 遇到 DTD 中元素属性声明时，将调用 AttlistDeclHandler 方法。其原型如下。

```
AttlistDeclHandler(elname, attname, type, default, required)
```

其参数含义如下。

- elname　　元素名。
- attname　　属性名。
- type　　元素数据类型。
- default　　属性默认值。
- required　　如果属性为必须的，则 required 为真。

当 XMLParser 遇到元素开始标记时，将调用 StartElementHandler 方法；当遇到结束标记时，将调用 EndElementHandler 方法。其原型如下。

```
StartElementHandler(name, attributes)
EndElementHandler(name)
```

其参数含义如下。

- name　　元素名。
- attributes　　元素属性。

当 XMLParser 遇到 XML 处理指令时，将调用 ProcessingInstructionHandler 方法。其原型如下。

```
ProcessingInstructionHandler(target, data)
```

其参数含义如下。

- target　　指令名称。
- data　　指令数据。

当 XMLParser 遇到字符数据时，将调用 CharacterDataHandler 方法。其原型如下。

```
CharacterDataHandler(data)
```

其参数含义如下。

- data　　字符数据。

当 XMLParser 遇到 XML 文档中的注释时，将调用 CommentHandler 方法。其原型如下。

```
CommentHandler(data)
```

其参数含义如下。

- data　　注释内容。

当 XMLParser 遇到 CDATA 开始时，将调用 StartCdataSectionHandler 方法；当遇到 CDATA 结束时，将调用 EndCdataSectionHandler 方法。

19.3.2 使用 xml.sax 处理 XML

与 xml.parsers.expat 模块不同，xml.sax 模块将分析器和处理器分离了。使用 xml.sax 模块时，可以使用 make_parser 函数创建分析器，它会返回一个 XMLReader 对象。然后使用 XMLReader 对象的 setContentHandler 方法设置 XMLReader 对象的 ContentHandler。在脚本中通过继承 ContentHandler 类，重载相应的处理方法，即可对 XML 文档进行处理。

如果需要对 DTD 进行处理，则可以使用 XMLReader 对象的 setDTDHandler 方法，设置 XMLReader 对象的 DTDHandler 句柄。

xml.sax 的分析器在分析 XML 文档时，如果是在 XML 文档开始时，则将调用 ContentHandler 的 startDocument 方法；如果是在 XML 文档结束时，则将调用 ContentHandler 的 endDocument 方法。其原型如下。

```
startElement(name, attrs)
endElement(name)
```

其参数含义如下。

- name　　元素名。
- attrs　　元素属性。

如果在 XML 文档中使用了命名空间，则将调用 startElementNS 方法和 endElementNS 方法。其原型如下。

```
startElementNS(name, qname, attrs)
endElementNS(name, qname)
```

其参数含义如下。

- name　　由 URI 和本地名组成的元组。

- qname　　元素名。
- attrs　　元素属性。

当 xml.sax 的分析器遇到字符数据时，将调用 ContentHandler 的 characters 方法。其原型如下。

```
characters(content)
```

其参数含义如下。

- content　　字符数据内容。

当 xml.sax 的分析器遇到 XML 处理指令时，将调用 ContentHandler 的 processingInstruction 方法。其原型如下。

```
processingInstruction(target, data)
```

其参数含义如下。

- target　　指令名称。
- data　　指令数据。

当 xml.sax 的分析器处理 DTD 时，将调用 DTDHandler 中的方法。在 DTDHandler 中主要提供了用于处理声明的 notationDecl 方法和用于处理分分析实体声明的 unparsedEntityDecl 方法。其原型如下。

```
notationDecl(name, publicId, systemId)
unparsedEntityDecl(name, publicId, systemId, ndata)
```

19.3.3 使用 xml.dom 处理 XML

DOM（Document Object Model），即文档对象模型，使用树式结构创建 XML 文档。Python 的 xml.dom 模块提供的 DOM 接口可以将 XML 文档转为树结构。Python 提供了基本的 DOM 分析系统 minidom 模块和复杂的 DOM 分析系统 pulldom 模块。minidom 模块适用于较小的 XML 文档，因为它将整个文档读到内存中。而 pulldom 模块则适用于较大的 XML 文档，它不会将整个 XML 文档一次性读入内存，从而减小内存的开销。

Python 的 xml.dom 模块中的接口较多，其常用的 Element 对象中有以下几种方法。

- getElementsByTagName()　　用于获取标记的分支。
- getElementsByTagNameNS()　　用于获取标记的分支（使用了命名空间的情况）。
- getAttribute()　　用于获取属性值。
- getAttributeNode()　　用于获取属性节点。
- getAttributeNS()　　用于获取属性值（使用了命名空间的情况）。
- getAttributeNodeNS()　　用于获取属性节点（使用了命名空间的情况）。
- removeAttribute()　　用于移除属性。
- removeAttributeNode()　　用于移除属性节点。

- removeAttributeNS() 用于移除属性（使用了命名空间的情况）。
- setAttribute() 用于设置属性。
- setAttributeNS() 用于设置属性（使用了命名空间的情况）。
- setAttributeNode() 用于设置属性节点。
- setAttributeNodeNS() 用于设置属性节点（使用了命名空间的情况）。

使用 minidom 模块时，应首先使用 parse 方法或 parseString 方法获取 XML 数据。获取 XML 数据后，就可以使用 xml.dom 中的方法对 XML 文档进行处理了。

19.4 简单的 RSS 阅读器

RSS（Really Simple Syndication）是一种描述和同步网站内容的格式，是一种信息聚合的技术，它提供了一种更为方便、高效的互联网信息的发布和共享，能用更少的时间分享更多的信息。RSS 是基于 XML 的，因此，可以使用 Python 处理 XML 的模块，做一个简单的 RSS 阅读器。

RSS 有自己的标准，可以通过阅读其标准编写一个 Python 脚本来处理网站的 RSS。但鉴于 XML 的良好可读性和良好的结构，只要找到一个简单的 RSS 文件，就完全可以写出一个简单的 RSS 阅读器。

下面以 Python 官方网站提供的 RSS 为例，先来看看 RSS 的内容。在 IE 浏览器中输入网址：http://www.python.org/channews.rdf，可以看到如下内容。

```
<?xml version="1.0" encoding="UTF-8"?>

<rss version="2.0">
  <channel>
    <title>Python News</title>
    <link>http://www.python.org/</link>
    <description>Python-related news and announcements.
      Python is an interpreted, interactive, object-oriented
      programming language. </description>

    <image>
      <title>Python logo</title>
      <url>http://www.python.org/images/python-logo.gif</url>
      <link>http://www.python.org/</link>
    </image>


      <item>
     <guid>http://www.python.org/news/index.html#Tue19November201308000100</guid>
     <title>Python 3.3.3 released</title>
     <description><![CDATA[<!--utf-8--><!--0.7-->



<p><a class="reference external" href="/download/releases/3.3.3/">Python
3.3.3</a> is now available.</p>]]></description>
     <pubDate>Tue, 19 November 2013, 08:00 +0100</pubDate>
      </item>
      <item>
```

```
    <guid>http://www.python.org/news/index.html#Tue12October201308000100</guid>
    <title>Python 3.3.3 release candidate 2 has been released</title>
    <description><![CDATA[<!--utf-8--><!--0.7-->


<p>The <a class="reference external" href="/download/releases/3.3.3/">second
release candidate for Python 3.3.3</a> has been released.</p>]]></description>
    <pubDate>Tue, 12 October 2013, 08:00 +0100</pubDate>
     </item>

     (此处省略部分内容)

     <item>
    <guid>http://www.python.org/news/index.html#Fri25January201321000200</guid
>
    <title>Python wiki server online again after recovery</title>
    <description><![CDATA[<!--utf-8--><!--0.7-->


<p>The Python wiki server is back online, after an attack on January 5 2013. All
passwords were reset, so you will have to use the password recovery function to
get a new password. Please see the <a class="reference external"
href="http://wiki.python.org/moin/WikiAttack2013">wiki attack description
page</a> for more details.</p>]]></description>
    <pubDate>Fri, 25 January 2013, 21:00 +0200</pubDate>
     </item>

  </channel>
</rss>
    省略部分内容
  </channel>
</rss>
```

可以看到，每一条主要的消息都处于“item”元素之间。在“item”元素中依次包含“guid”、“title”、“description”和“pubDate”元素。其中，“title”元素包含了消息的标题，“description”元素包含了消息的简要描述，“pubDate”元素包含了消息发布的时间。Python 官方的 RSS 文件非常直观，通过查看 RSS 文件内容即可了解其结构。下面所示的 pyRSS.py 脚本仅对“title”和“pubDate”元素进行了处理，获取其字符数据，即获取消息的标题和发布时间。

根据对上述 XML 文档的分析，可编写出下面所示的 RSS 阅读器脚本 pyRSS.py。

```
# -*- coding:utf-8 -*-
# file: pyRSS.py
#
import tkinter
import urllib.request
import xml.parsers.expat
class MyXML:                                                    # XML 解析类
    def __init__(self, edit):
        self.parser = xml.parsers.expat.ParserCreate()          # 生成 XMLParser
        self.parser.StartElementHandler = self.start            # 起始标记处理方法
        self.parser.EndElementHandler = self.end                # 结束标记处理方法
        self.parser.CharacterDataHandler = self.data            # 字符数据处理方法
```

```
        self.title = False                                  # 状态标志
        self.date = False
        self.edit = edit                                    # 多行文本框对象
    def start(self, name, attrs):                           # 起始标记处理方法
        if name == 'title':                                 # 判断是否为 title 元素
            self.title = True                               # 标志设为真
        elif name == 'pubDate':                             # 判断是否为 pubDate
            self.date = True                                # 标志设为真
        else:
            pass
    def end(self, name):                                    # 结束标记处理
        if name == 'title':
            self.title = False                              # 标志设为假
        elif name == 'pubDate':
            self.date = False                               # 标志设为假
        else:
            pass
    def data(self,data):                                    # 字符数据处理方法
        if self.title:                                      # 根据标志状态输出数据
            self.edit.insert(tkinter.END,
                  '*****************************\n')
            self.edit.insert(tkinter.END, 'Title: ')
            self.edit.insert(tkinter.END, data + '\n')
        elif self.date:
            self.edit.insert(tkinter.END, 'Date: ')
            self.edit.insert(tkinter.END, data + '\n')
        else:
            pass
    def feed(self, data):
        self.parser.Parse(data, 0)
class Window:
    def __init__(self, root):
        self.root = root                                    # 创建组件
        self.get = tkinter.Button(root,
                text = '获取 RSS', command = self.Get)
        self.get.place(x = 280, y = 15)
        self.frame = tkinter.Frame(root, bd=2)
        self.scrollbar = tkinter.Scrollbar(self.frame)
        self.edit = tkinter.Text(self.frame,yscrollcommand = self.scrollbar.set,
                width = 96, height = 32)
        self.scrollbar.config(command=self.edit.yview)
        self.edit.pack(side = tkinter.LEFT)
        self.scrollbar.pack(side=tkinter.RIGHT, fill=tkinter.Y)
        self.frame.place(y = 50)
    def Get(self):
        url = 'http://www.python.org/channews.rdf'
        page = urllib.request.urlopen(url)                  # 打开 URL
        data = page.read()                                  # 读取 URL 内容
        parser = MyXML(self.edit)                           # 生成实例对象
        parser.feed(data)                                   # 处理 XML 数据
root = tkinter.Tk()
window = Window(root)
root.minsize(600,480)
root.maxsize(600,480)
```

```
root.mainloop()
```

运行 pyRSS.py 脚本后，单击【获取 RSS】按钮，将显示如图 19-5 所示的结果，在窗口的文本框中列出了消息的标题和发布日期等信息。



图 19-5　单击【获取 RSS】按钮后的结果

19.5　本章小结

本章介绍了如何处理从网络上获取的 HTML 和 XML 文件。首先介绍了 HTMLParse 类的基本用法，接着以两个实际案例演示了使用 HTMLParese 类处理 HTML 的方法。然后介绍了 XML 的基本概念和用 xml 模块解析 XML 文件的操作。最后综合应用处理 XML 的方法编写了一个 RSS 阅读器。

下一章将进入另一个主题：正则表达式。

◆　　◆　　◆

第 20 章　功能强大的正则表达式

本章包括

- 正则表达式的基本元字符
- 使用 re 模块处理正则表达式
- 用正则表达式对象提速
- 匹配和搜索的结果对象：Match 对象
- 常用正则表达式分析
- 编译生成正则表达式对象
- 正则表达式中的分组
- 使用正则表达式处理文件

正则表达式是用某种模式去匹配一类具有共同特征的字符串。正则表达式主要用于处理文本，正则表达式能够使文本处理变得简单，尤其对于复杂的查找和替换这样的工作，使用正则表达式会非常快地完成。流行的文本编辑器（如 Emacs、Vim 等）大都支持正则表达式。

20.1　正则表达式概述

在 Python 中，主要使用 re 模块进行正则表达式的操作。re 模块提供了 Perl 风格的正则表达式，使正则表达式具有更好的可读性，以及更强的功能。

20.1.1　正则表达式的基本元字符

元字符是正则表达式中具有特定含义的字符，在正则表达式中，可以在字符串中使用元字符来匹配字符串的各种可能的情况。常用的元字符如表 20-1 所示。

表 20-1　元字符表

元字符	含义
.	匹配除换行符以外的任意单个字符，如 "r.d" 会匹配 "red"、"r d" 等，但不会匹配 "read"
*	匹配位于*之前的 0 个或多个字符，如 "r*ed" 会匹配 "ed"、"rred"、"rrred"、"red" 等
+	匹配位于+之前的一个或多个字符，如 "r+ed" 会匹配 "rred"、"rrred"，但不会匹配 "ed"
\|	匹配位于\|之前或者之后的字符，如 "red\|blue" 会匹配 "red"、"blue"
^	匹配行首
$	匹配行尾
?	匹配位于? 之前的零个或一个字符，如 "r?ed" 会匹配 "ed"、"red" 等，但不会匹配 "rrred"
\	表示位于\之后的为转义字符
[]	匹配位于[]中的任意一个字符，如 r[ae]d 会匹配 "rad"、"red" 等
()	将位于()内的内容当作一个整体
{}	按{}中的次数进行匹配

元字符还可以结合起来使用。".*" 可以匹配任意个字符，如 "r.*d" 可以匹配 "rd"、"red"、"read" 等。".+" 可以匹配任意的一个或者多个字符，如 "r.+d" 可以匹配 "red"、"read"，但

不会匹配“rd”。“.?”可以匹配任意的零个或一个字符，如“r.?d”可以匹配“rd”、“red”，但不会匹配“read”。

“^”匹配行首，对于下面所示的一段文字，“^red”只会匹配文中的第三个“red”。而“red$”，则只会匹配文中的第二个“red”。

```
a red hat
blue and red
red and blue
```

在“[]”中，还可以使用“-”来表示某一范围。例如，在“[]”中，“[a-z]”表示从“a”到“z”的所有小写字母，同样，“[A-Z]”表示从“A”到“Z”的所有大写字母，而“[0-9]”表示从“0”到“9”的数字，“[a-zA-Z0-9]”表示任意的字母或者数字。

20.1.2 常用正则表达式分析

使用正则表达式可以简化程序设计。例如，在文本文件中包含一些联系人的手机号码，如果需要将使用联通和移动号码的人区分出来，则可以使用正则表达式分别匹配。对于联通的手机号，第3位是0、1、2。而移动的则是4~9中的某一个数字，可以据此来判断。下面的正则表达式能够匹配联通的手机号。

```
13[0-2][0-9]{8}
```

其中13表示匹配手机号的前两位。[0-2]匹配手机号的第3位，表示0~2之间的任意一个数字。[0-9]{8}组合起来匹配手机号剩下的8位。[0-9]表示0~9之间的任意一个数字，{8}表示匹配[0-9]八次，也就是匹配任意一个8位数。下面的正则表达式能够匹配移动的手机号。

```
13[4-9][0-9]{8}
```

其与匹配联通的手机号唯一不同的地方是第3位是用[4-9]。[4-9]表示4~9之间的任意一个数字。



以上只是处理以13开头的手机号码，随着手机号码的不断推出，现在还有14、15、18开头的手机号。这里为了简化正则表达式，使初学者能看懂，暂不处理其他号段的情况。

同样，如果联系人的信息中还包含邮政编码，而又需要将其按地区来区分的话，则也可以使用正则表达式来处理。对于邮政编码的匹配要相对简单一些，例如，北京的邮政编码前3位为100。采用与匹配手机号同样的方式，只需在100之后匹配任意一个3位数即可。

```
100[0-9]{3}
```

使用正则表达式匹配数字十分方便，但如果匹配字符串则需要考虑较多的情况。例如，需要找出某一文件中所有的网址，则较为复杂。以“http://www.python.org”为例，这个网址可以分成四部分，首先是“http://”，然后是“www”，再就是站名“python”，剩下的是后缀“org”。可能有的网址比较完整，具有以上4个部分，而有些网址则不规范（如不包含“http://”部分）。

将“http://www”当作一个部分，其可能的情况为“http://www”或者“www”，这一部分的正

则表达式匹配可以写成如下所示。

```
(http://www|www)     # 使用 "()" 表示其为一个整体，使用 "|" 表示其中任何一个满足则匹配
```

中间的站名作为一部分，这部分可能是字母、数字或者 "-"，因此该部分的正则表达式匹配可以写成如下形式。

```
[a-z0-9-]*           # [a-z0-9-]表示字母、数字或者 "-"，"*" 表示匹配 0 个或多个前边的字符
```

剩下的后缀，考虑到可能为两个或三个的字符，因此写成如下形式。

```
[a-z]{2,3}           # {2,3}表示重复两次或三次，即匹配由两个或三个字母组成的字符串
```

由于其中还包含 "."，而在正则表达式中其具有特殊含义，需要将其使用 "\" 转义。因此，完整的正则表达式如下。

```
(http://www|www)\.[a-z0-9-]*\.[a-z]{2,3}
```

此处没有考虑诸如 "com.cn" 这样的形式，读者可以试着自己将其完成。

20.2 支持正则表达式的 re 模块

Python 中的 re 模块提供了对正则表达式的支持。虽然 Python 中有一个 string 模块用来对字符串进行处理，但 string 模块只能完成简单的操作，而使用 re 模块则可以完成对复杂字符串的操作。re 模块提供了以下几类对字符串进行操作的函数。

20.2.1 用 match 函数进行搜索

re.match()函数用于在字符串中匹配正则表达式，如果匹配成功则返回 MatchObject 对象实例。

re.search()函数用于在字符串中查找正则表达式，如果找到则返回 MatchObject 对象实例。

re.findall()函数用于在字符串中查找所有符合正则表达式的字符串，并返回这些字符串的列表。如果在正则表达式中使用了组，则返回一个元组。

re.match()函数和 re.search()函数的作用基本一样。不同的是，re.match()函数只从字符串中第一个字符开始匹配，而 re.search()函数则搜索整个字符串。以上 3 个函数的原型如下。

```
re.match( pattern, string[, flags])
re.search( pattern, string[, flags])
findall( pattern, string[, flags])
```

其参数含义相同，具体如下。

- pattern　　匹配模式。
- string　　要进行匹配的字符串。
- flags　　可选参数，进行匹配的标志。

参数 flags 可以是以下选项。

- re.I　　忽略大小写。
- re.L　　根据本地设置而更改\w、\W、\b、\B、\s 以及\S 的匹配内容。
- re.M　　多行匹配模式。
- re.S　　使 "." 元字符也匹配换行符。
- re.U　　匹配 Unicode 字符。
- re.X　　忽略 pattern 中的空格，并且可以使用 "#" 注释。

上述的几个匹配标志可以同时使用，同时使用几个编译标志时，需要使用 "|" 对共用的编译标志进行运算。以下代码是使用上述函数进行匹配和搜索。

```
>>> import re                                    # 导入 re 模块
>>> s = 'Life can be good'                       # 定义字符串
>>> print(re.match('can',s))                     # 在字符串中匹配 "can"
None                                             # 输出为 None，表示未找到
>>> print(re.search('can',s))                    # 在字符串中搜索 "can"
<_sre.SRE_Match object at 0x010DAB48>            # 返回一个 Match object，表示找到
>>> print(re.match('l.*',s))                     # 匹配任意以字母 "l" 开头的字符串
None                                             # 表示未找到
>>> print(re.match('l.*',s,re.I))                # 此处设置忽略大小写
<_sre.SRE_Match object at 0x010DAC28>            # 返回一个 Match object，表示找到
>>> re.findall('[a-z]{3}',s)                     # 查找所有 3 个字母的小写字符串
['ife', 'can', 'goo']
>>> re.findall('[a-z]{1,3}',s)                   # 查找所有由 1 到 3 个字母组成的小写字符串
['ife', 'can', 'be', 'goo', 'd']
```

20.2.2 用 sub 函数进行内容替换

re.sub()函数用于替换在字符串中符合正则表达式的内容，它返回替换后的字符串。re.subn()函数与 re.sub()函数相同，只不过 re.subn()函数将返回一个元组，用来保存替换的结果和替换次数。其函数原型如下。

```
re.sub( pattern, repl, string[, count])
re.subn( pattern, repl, string[, count])
```

其参数含义如下。

- pattern　　正则表达式模式。
- repl　　要替换成的内容。
- string　　进行内容替换的字符串。
- count　　可选参数，最大替换次数。

以下代码是使用上述函数进行内容替换。

```
>>> import re                                    # 导入 re 模块
>>> s = 'Life can be bad'                        # 定义字符串
>>> re.sub('bad','good',s)                       # 用 "good" 替换 "bad"
'Life can be good'
```

```
>>> re.sub('bad|be','good',s)              # 用 "good" 替换 "bad" 或者 "be"
'Life can good good'
>>> re.sub('bad|be','good',s,1)            # 用 "good" 替换 "bad" 或者 "be"，但只替换一次
'Life can good bad'
>>> re.subn('bad|be','good',s,1)           # 用 "good" 替换 "bad" 或者 "be"，但只替换一次
('Life can good bad', 1)                   # 返回由替换后的字符串和替换的次数组成的元组
>>> r = re.subn('bad|be','good',s)         # 用 "good" 替换 "bad" 或者 "be"
>>> print(r[0])                            # 输出元组第一项
Life can good good
>>> print(r[1])                            # 输出元组第二项
2
```

20.2.3 用 split 函数分割字符串

re.split()函数用于分割字符串，它返回分割后的字符串列表。其函数原型如下。

```
re.split( pattern, string[, maxsplit = 0])
```

其参数含义如下所示。

- pattern　　正则表达式模式。
- string　　要分割的字符串。
- maxsplit　　可选参数，最大分割次数。

以下代码是使用上述函数对字符串进行分割操作。

```
>>> import re                              # 导入 re 模块
>>> s = 'Life can be bad'                  # 定义字符串
>>> re.split(' ',s)                        # 使用空格分割字符串( 注意：单引号之间有一个空格 )
['Life', 'can', 'be', 'bad']
>>> r = re.split(' ',s,1)                  # 只分割一次
>>> for i in r:                            # 遍历分割后返回的列表
... print(i)
...
Life
can be bad
>>> re.split('b',s)                        # 使用字母 "b" 分割字符
['Life can ', 'e ', 'ad']
```

20.3 编译生成正则表达式对象

使用 re.compile()函数将正则表达式编译生成正则表达式对象实例后，可以使用正则表达式对象实例提供的属性和方法对字符串进行处理。

20.3.1 以“\”开头的元字符

除了基本的元字符外，还有一类以“\”开头的元字符。以“\”开头的元字符主要表示某一类型的集合，如数字的集合、字母的集合等。常用的以“\”开头的元字符如表 20-2 所示。

表 20-2 "\" 开头的元字符

转义字符	含义
\b	匹配单词头或单词尾
\B	与\b 含义相反
\d	匹配任何数字
\D	匹配任何非数字
\s	匹配任何空白字符
\S	匹配任何非空白字符
\w	匹配任何字母、数字以及下画线
\W	匹配任何非字母、数字以及下画线

以 "\" 开头的元字符也可以与其他的元字符配合使用。例如，"\d*" 可以匹配任意由数字组成的字符串，其中，"\d" 相当于[0-9]，"\w" 相当于[a-zA-Z0-9_]。以下代码演示了如何在正则表达式中使用以 "\" 开头的元字符。

```
>>> import re                                    # 导入 re 模块
>>> s = 'Python can run on Windows'              # 定义字符串
>>> re.findall('\\bo.+?\\b',s)                   # 查找首字母为 "o" 的单词
['on']
>>> re.findall('\\Bo.+?',s)                  # 查找含字母 "o" 的单词，但 "o" 不是单词的首字母
['on', 'ow']
>>> re.findall('\so.+?',s)                       # 使用空字符来匹配首字母为 "o" 的单词
[' on']                                          # 返回的字符串中含有一个空格
>>> re.findall('\\b\w.+?\\b',s)                  # 查找字符串中的所有单词
['Python', 'can', 'run', 'on', 'Windows']
>>> re.findall('\d\.\d','Python 2.5')            # 查找 x.x 的数字形式
['2.5']                                          # 此处匹配 2.5
>>> re.findall('\D+','Python 2.5')               # 查找不含数字的字符
['Python ', '.']
>>> re.split('\s',s)                             # 使用空字符分割字符串
['Python', 'can', 'run', 'on', 'Windows']
>>> re.split('\s',s,1)                           # 使用空字符分割字符串，但只分割一次
['Python', 'can run on Windows']
>>> re.findall('\d\w+?','abc3de')                # 查找以数字开始的字符
['3d']
```

在上述例子中，是使用'\\bo.+?\\b'来匹配首字母为 "o" 的单词，而不是直接使用'\\b\o.+\\b'或者'\\b\o.*\\b'来匹配。如果使用'\\b\o.+\\b'或者'\\b\o.*\\b'，则输出如下。

```
>>> re.findall('\\b\o.+\\b',s)
['on Windows']
>>> re.findall('\\b\o.*\\b',s)
['on Windows']
```

可以看到，本来预想的是 "on" 后有一个空格，应该只匹配到 "on"。然而最终结果连之后的 "Windows" 也匹配了。这是因为 "+"、"*" 等元字符为 "贪婪" 模式的元字符，它们尽可能匹配更多的字符。为了避免 "+"、"*" 等元字符过多的匹配，可以在其之后使用 "非贪婪" 模式的

"?"，或者也可以使用"{}"指定匹配的次数。

20.3.2 用 compile 函数编译正则表达式

re 模块中包含一个 re.compile()函数，可以使用 re.compile()函数将正则表达式编译生成一个 RegexObject 对象实例。然后通过生成的 RegexObject 对象实例对字符串进行操作，如查找、替换等。re.compile()的函数原型如下。

```
compile( pattern[, flags])
```

其参数含义如下。

- pattern　　正则表达式的匹配模式。
- flags　　可选参数，编译标志。

以下代码编译生成一个 RegexObject 对象实例。

```
>>> import re                                    # 导入 re 模块
>>> re.compile('a*b',re.I|re.X)                  # 编译正则表达式，忽略大小写和模式中的空格
<_sre.SRE_Pattern object at 0x011232F0>
# 使用 re.X 编译标志，表示在匹配模式中忽略注释及空格等字符
>>> re.compile('''
... \b      # 匹配单词开始
... AA?     # 以 A 或 AA 开头
... \d      # 匹配一个数字 i
... \w*     # 匹配任意字符
...         # 一个空行
... \b      # 匹配单词结束
... ''',re.X)
<_sre.SRE_Pattern object at 0x01093BD8>
```

20.3.3 在正则表达式中使用原始字符串

原始字符串在第 3 章中已经提及。原始字符串是为正则表达式设计的，以提高正则表达式的可读性，减少"\"在正则表达式中的数目。

由于在正则表达式中，是使用以"\"开头的字符来表示某些特殊的含义，而在字符串中，转义字符也是以"\"开头的，这就导致了冲突。例如，在正则表达式中，"\b"表示匹配一个单词的开始或者结束，而在字符串中，"\b"则表示退格。如果在正则表达式中使用"\b"，则应该写成"\\b"。在 re.compile()中，"\b"的正确写法如下。

```
re.compile('\\ba.?')
```

如果使用原始字符串，则写法如下。

```
re.compile(r'\ba.?')
```

如果要在正则表达式中匹配一个以"\"开头的字符串，如"\word"，则首先应将"\"转义，以避免将其写成"\\word"。而"\\word"包含有两个"\"字符串，因此又需要两个"\"将其转义，

因此正确的写法如下。

```
re.compile('\\\\word')
```

而如果使用原始字符串，则正确的写法如下。

```
re.compile(r'\\word')
```

20.4 用正则表达式对象提速

正则表达式对象提供了与 re 模块中函数类似的字符串操作方法，不过，使用正则表达式对象提供的这些函数，可提高处理的速度。常用正则表达式对象的属性和方法可分为以下几种。

20.4.1 使用 match 方法匹配和搜索

正则表达式对象的 match()方法用于从字符串开始处进行匹配，或者从指定位置处进行匹配。要匹配的字符串必须位于开头或者参数指定的位置才会匹配成功。其原型如下。

```
match( string[, pos[, endpos]])
```

其参数含义如下。

- ◆ string　　要进行匹配的字符串。
- ◆ pos　　可选参数，进行匹配的起始位置。
- ◆ endpos　　可选参数，进行匹配的结束位置。

如果匹配成功，则 match()会返回一个 MatchObject 对象实例。

与 match()类似，search()方法也可对字符串进行查找，不同的是，search()方法是在整个字符串中搜索。如果查找成功，则 search()将返回一个 MatchObject 对象实例。其原型如下。

```
search( string[, pos[, endpos]])
```

其参数含义如下。

- ◆ string　　要进行匹配的字符串。
- ◆ pos　　可选参数，进行查找的起始位置。
- ◆ endpos　　可选参数，进行查找的结束位置。

正则表达式对象的 findall()方法用于在字符串中查找所有符合正则表达式的字符串，并返回这些字符串的列表。如果在正则表达式中使用了组，则返回一个元组。其原型如下。

```
findall( string[, pos[, endpos]])
```

其参数含义与 search()方法相同。

以下代码是使用 RegexObject 对象对字符串进行匹配和搜索。

```
# 导入 re 模块
```

```
>>> import re
# 编译正则表达式，"go*d" 表示在 "g" 和 "d" 之间有任意个字母 "o" 的单词，如 "gd"，"god"，"good"
>>> r = re.compile('go*d')
# 在字符串开始处匹配，若没有返回值，则表示匹配失败
>>> r.match('Life can be good')
# 从字符串的第 13 个字符开始匹配（字符串从 0 开始），也就是从字母 "g" 开始，返回 MatchObject 对象实例
>>> r.match('Life can be good',12)
<_sre.SRE_Match object at 0x01122640>
# 在字符串中搜索 "go*d"，返回 MatchObject 对象实例，表示字符串中含有 "go*d"
>>> r.search('Life can be good')
<_sre.SRE_Match object at 0x011226E8>
# 重新编译，匹配字母 "b" 和字母 "g" 之间包含一个字母以及一个空字符的情况
>>> r = re.compile('b.\sg')
# 在字符串中搜索，此处匹配的是 "be g"
>>> r.search('Life can be good')
<_sre.SRE_Match object at 0x01122640>
# 重新编译，匹配任意两个字母后跟一个空字符和字母 "g" 的情况
>>> r = re.compile('\w.\sg')
# 在字符串中搜索，此处匹配的是 "be g"
>>> r.search('Life can be good')
<_sre.SRE_Match object at 0x011227C8>
# 匹配后边有一个空字符的任意包含两个或者三个字符的单词
>>> r = re.compile('\\b\w..?\s')
# 使用 findall()方法查找
>>> r.findall('Life can be good')
['can ', 'be ']
```

20.4.2 使用 sub 方法替换内容

正则表达式对象的 sub()和 subn()方法用于对字符串的替换，其原型如下。

```
sub( repl, string[, count = 0])
subn( repl, string[, count = 0])
```

其参数含义相同，具体如下。

- repl　　要替换成的内容。
- string　　进行内容替换的字符串。
- count　　可选参数，最大替换次数。

以下代码是将所有以字母 "b" 开头的单词替换成 "*"。

```
>>> import re                          # 导入 re 模块
>>> s = '''Life can be good;           # 定义字符串
... Life can be bad;
... Life is mostly cheerful;
... But sometimes sad.'''
>>> r = re.compile('b\w*',re.I)        # 编译正则表达式，忽略大小写
>>> new = r.sub('*',s)                 # 使用 sub()替换字符
>>> print(new)                         # 输出结果，可以看到所有以 "b" 开头的单词都被替换
Life can * good;
```

```
Life can * *;
Life is mostly cheerful;
* sometimes sad.
>>> new = r.sub('*',s,2)                    # 只在字符串中替换两次
>>> print(new)
Life can * good;
Life can * bad;
Life is mostly cheerful;
But sometimes sad.
>>> r = re.compile('b\w*')                  # 重新编译，不忽略大小写
>>> new = r.subn('*',s)                     # 使用 subn()替换字符，它返回一个元组
>>> print(new[0])                           # 输出替换后的字符串，可以看到 "But" 没有被替换
Life can * good;
Life can * *;
Life is mostly cheerful;
But sometimes sad.
>>> print(new[1])                           # 输出替换的次数
3
>>> new = r.subn('*',s,1)                   # 只在字符串中替换一次
>>> print(new[0])
Life can * good;
Life can be bad;
Life is mostly cheerful;
But sometimes sad.
>>> print(new[1])
1
```

20.4.3 使用 split 方法分割字符串

正则表达式对象的 split()方法用于对字符串进行分割。其原型如下。

```
split( string[, maxsplit = 0])
```

其参数含义如下。

- ◆ string　　要分割的字符串。
- ◆ maxsplit　　可选参数，最大分割次数。

以下代码是使用 split()方法对字符串进行分割。

```
>>> import re                               # 导入 re 模块
>>> s = '''Life can be good;                # 定义字符串
... Life can be bad;
... Life is mostly cheerful;
... But sometimes sad.'''
>>> r = re.compile('\s')                    # 编译匹配空字符的正则表达式
>>> news = r.split(s)                       # 以空字符分割字符串
>>> print(news)                             # 返回一个列表
['Life', 'can', 'be', 'good;', 'Life', 'can', 'be', 'bad;', 'Life', 'is', 'mostly',
'cheerful;', 'But', 'sometimes', 'sad.']
>>> news = r.split(s,4)                     # 只分割四次
>>> for new in news:                        # 遍历列表，输出分割后的字符串
... print(new)
...
```

```
Life
can
be
good;
Life can be bad;
Life is mostly cheerful;
But sometimes sad.
>>> r = re.compile('b\w*',re.I)      # 编译匹配以字母 "b" 开头的字符串，忽略大小写
>>> news = r.split(s)                # 分割字符串返回列表
>>> print(news)                      # 输出列表
['Life can ', ' good;\nLife can ', ' ', ';\nLife is mostly cheerful;\n', ' sometimes
sad.']
>>> news = r.split(s,1)              # 只分割一次
>>> for new in news:                 # 遍历列表，输出字符串
... print(new)
...
Life can
 good;
Life can be bad;
Life is mostly cheerful;
But sometimes sad.
>>> r = re.compile('\w*e',re.I)      # 编译匹配以字母 "e" 结尾的字符串，忽略大小写
>>> news = r.split(s)
>>> print(news)
['', ' can ', ' good;\n', ' can ', ' bad;\n', ' is mostly ', 'rful;\nBut ', 's sad.']
```

20.5 正则表达式中的分组

在正则表达式中使用组，可以将正则表达式分解成几个不同的组成部分。在完成匹配或者搜索后，可以使用分组编号访问不同部分的匹配内容。

20.5.1 分组的概述

在正则表达式中，以一对圆括号 "()" 来表示位于其中的内容属于一个分组。例如，"(re)+" 可匹配 "rere"、"rerere" 等多个 "re" 重复的情况。分组在匹配由不同部分组成的一个整体时非常有用。如电话号码由区号和号码组成，在正则表达式中可以使用两个分组来进行匹配：一个分组匹配区号，另一分组匹配后边的号码。代码如下。

```
>>> import re
>>> s = 'Phone No. 010-87654321'
>>> r = re.compile(r'(\d+)-(\d+)')
>>> m = r.search(s)
>>> m
<_sre.SRE_Match object at 0x01107E30>
>>> m.group(1)
'010'
>>> m.group(2)
'87654321'
>>> m.groups()
('010', '87654321')
```

在正则表达式中，可以通过使用 "(?P<组名>)" 为组设置一个名字，通过使用如下所示模式，

将第一个组的名字设置为“Area”，将第二组的名字设置为“No”。

```
r'(?P<Area>\d+)-(?P<No>\d+)'
```

上述的例子可以修改成如下所示。

```
>>> import re
>>> s = 'Phone No. 010-87654321'
>>> r = re.compile(r'(?P<Area>\d+)-(?P<No>\d+)')
>>> m = r.search(s)
>>> m
<_sre.SRE_Match object at 0x01122BA8>
>>> m.groupdict()
{'Area': '010', 'No': '87654321'}
>>> m.group('No')
'87654321'
>>> m.group('Area')
'010'
```

20.5.2 分组的扩展语法

除了在组中使用“(?P<>)”来命名组名外，还可以使用以下几种以“?”开头的扩展语法。

- (?iLmsux) 设置匹配标志，可以是 i，L，m，s，u，x 以及它们的组合。其含义与编译标志相同。
- (?:...) 匹配但不捕获该匹配的子表达式，即它是一个非捕获匹配，不存储供以后使用的匹配。
- (?P=name) 表示在此之前的名为 name 的组。
- (?#...) 表示注释。
- (?=...) 用于正则表达式之后，表示如果“=”后的内容在字符串中出现则匹配，但不返回“=”后的内容。
- (?!...) 用于正则表达式之后，表示如果“!”后的内容在字符串中不出现则匹配，但不返回“!”后的内容。
- (?<=...) 用于正则表达式之前，与(?=...)含义相同。
- (?<!...) 用于正则表达式之前，与(?!...)含义相同。

上述模式的使用如下所示。

```
>>> import re                                  # 导入 re 模块
>>> s = '''Life can be good;                   # 定义字符串
... Life can be bad;
... Life is mostly cheerful;
... But sometimes sad.
... '''
>>> r = re.compile(r'be(?=\sgood)')            # 编译正则表达式，只匹配其后单词为"good"的"be"
>>> m = r.search(s)                            # 搜索字符串
>>> m                                          # 查看 m
<_sre.SRE_Match object at 0x0111ED40>          # 返回一个 Matchobject 对象实例，表示查找到单词
>>> m.span()                                   # 输出匹配到的单词在字符串中的位置
```

```
(9, 11)
>>> r.findall(s)                          # 使用 findall()方法输出所有匹配的单词
['be']
>>> r = re.compile('be')                  # 重新编译正则表达式，匹配单词 "be"
>>> r.findall(s)                          # 使用 findall()方法输出所有匹配的单词
['be', 'be']
>>> r = re.compile(r'be(?!\sgood)')       # 匹配之后单词不为 "good" 的 "be"
>>> m = r.search(s)                       # 搜索字符串
>>> m
<_sre.SRE_Match object at 0x010DAAD8>  # 返回一个 Matchobject 对象实例，表示查找到单词
>>> m.span()                              # 输出匹配到的单词在字符串中的位置
(27, 29)
>>> r = re.compile(r'(?:can\s)be(\sgood)')      # 使用组来匹配 "be good"
>>> m = r.search(s)
>>> m
<_sre.SRE_Match object at 0x01112660>
>>> m.groups()                            # 使用 groups()方法输出组
(' good',)
>>> m.group(1)                            # 使用组编号输出组
' good'
>>> r = re.compile(r'(?P<first>\w)(?P=first)') # 使用组名重复，此处匹配具有两个重复字母的单词
>>> r.findall(s)                          # 输出匹配到的字母
['o', 'e']
>>> r = re.compile(r'(?<=can\s)b\w*\b')     # 匹配以字母 "b" 开头位于 "can" 之后的单词
>>> r.findall(s)                          # 输出匹配到的单词
['be', 'be']
>>> r = re.compile(r'(?<!can\s)b\w*\b')# 匹配以字母 "b" 开头不位于 "can" 之后的单词
>>> r.findall(s)
['bad']
>>> r = re.compile(r'(?<!can\s)(?i)b\w*\b')     # 重新编译，忽略大小写
>>> r.findall(s)
['bad', 'But']
```

20.6 匹配和搜索的结果对象：Match 对象

Match 对象实例是由正则表达式对象的 match 方法及 search 方法在匹配成功后返回的。Match 对象有以下常用的方法和属性，用于对匹配成功的正则表达式进行处理。

20.6.1 使用 Match 对象处理组

group()、groups()和 groupdict()方法都是处理在正则表达式中使用“()”分组的情况。不同的是，group()的返回值为字符串，当传递多个参数时其返回值为元组；groups()的返回值为元组；groupdict()的返回值为字典，其原型分别如下。

```
group( [group1, ...])
groups( [default])
groupdict( [default])
```

对于 group()，其参数为分组的编号。如果向 group()传递多个参数，则其返回各个参数所对应

的字符串组成的元组。对于groups()和groupdict()，一般不需要向其传递参数。以下代码是使用上述三种方法对字符串进行操作。

```
>>> import re                                    # 导入 re 模块
>>> s = '''Life can be dreams,                  # 定义字符串
... Life can be great thoughts;
... Life can mean a person,
... Sitting in a court.'''
>>> r = re.compile('\\b(?P<first>\w+)a(\w+)\\b')
                                                # 编译正则表达式，匹配所有包含字母 "a" 的单词
>>> m = r.search(s)                             # 从头开始搜索，serach()返回搜索到的第一个单词
>>> m.groupdict()                               # 使用 groupdict()输出字典
{'first': 'c'}
>>> m.groups()                                  # 使用 groups()输出元组
('c', 'n')
>>> m = r.search(s,9)                           # 从指定位置开始重新搜索
>>> m.group()                                   # 输出匹配到的字符串
'dreams'
>>> m.group(1)                                  # 输出第一对圆括号中的内容，即字母 "a" 之前部分
'dre'
>>> m.group(2)                                  # 输出第二对圆括号中的内容，即字母 "a" 之后部分
'ms'
>>> m.group(1,2)                                # 全部输出，返回一个元组
('dre', 'ms')
>>> m.groupdict()                               # 使用 groupdict()输出字典
{'first': 'dre'}
>>> m.groups()                                  # 使用 groups()输出元组
('dre', 'ms')
```

20.6.2 使用 Match 对象处理索引

start()、end()和span()方法返回所匹配的子字符串的索引，其原型分别如下。

```
start([groupid=0])
end([groupid=0])
span([groupid=0])
```

其参数含义相同，groupid为可选参数，即分组编号。如果不向其传递参数，则返回整个子字符串的索引。start()方法返回子字符串或者组的起始位置索引。end()方法返回子字符串或者组的结束位置索引。而span()方法则以元组的形式返回以上两者。其使用方法如下。

```
>>> import re                                    # 导入 re 模块
>>> s = '''Life can be dreams,                  # 定义字符串
... Life can be great thoughts;
... Life can mean a person,
... Sitting in a court.'''
r = re.compile('\\b(?P<first>\w+)a(\w+)\\b')    # 编译正则表达式匹配含有字母 "a" 的单词
m = r.search(s,9)                               # 从字符串中第 10 个字符开始搜索
>>> m.start()                                   # 输出匹配到的子字符串的起始位置
12
>>> m.start(1)                                  # 输出第一组的起始位置
```

```
12
>>> m.start(2)                                    # 输出第二组的起始位置
16
>>> m.end(1)                                      # 输出第一组的子字符串的结束位置
15
>>> m.end()                                       # 输出子字符串的结束位置
18
>>> m.span()                                      # 输出子字符串的开始和结束位置
(12, 18)
>>> m.span(2)                                     # 输出第二组子字符串的开始和结束位置
(16, 18)
```

20.7 使用正则表达式处理文件

正则表达式是处理文本文件的强有力工具。本节将给出一个简单地使用正则表达式处理 Python 脚本中的函数和变量的例子。

在 Python 脚本中，函数定义必须以“def”开头，因此处理函数的过程相当简单。为了使代码更加简洁，此处假设脚本编写规范是在关键字“def”后跟一个空格，然后就是函数名，接着就是参数。不考虑使用多个空格的情况。

Python 脚本中的变量不好处理，因为变量一般不需要事先声明，往往都是直接赋值。因此在脚本中首先处理了变量直接赋值的情况。通过匹配单词后接“=”的情况查找变量名。同样，为了代码简洁，仅考虑比较规范整洁的写法，变量名与“=”之间有一空格。另外，还有一类变量是在 for 循环语句中直接使用的，因此脚本中又特别处理了 for 循环的情况。为了使代码简洁，脚本并没有处理变量名重复的情况。整个脚本的代码如下。

```
# -*- coding:utf-8 -*-
# file : GetFunction.py
#
import re
import sys
def DealWithFunc(s):
    r = re.compile(r'''
        (?<=def\s)      # 前边必须含有 def，且 def 后跟一个空格
        \w+             # 匹配函数明
        \(.*?\)         # 匹配参数
        (?=:)           # 后边必须跟一个 ":"
        ''',re.X | re.U)    # 设置编译选项，忽略模式中的注释
    return r.findall(s)
def DealWithVar(s):
    vars = []           # 定义一个列表，因为这里分两种情况处理
    r = re.compile(r'''
        \b              # 匹配单词开始
        \w+             # 匹配变量名
        (?=\s=)         # 处理为变量赋值的情况
        ''',re.X | re.U)
    vars.extend(r.findall(s))
    r = re.compile(r'''
        (?<=for\s)      # 处理变量位于 for 语句中的情况
```

```
        \w+                 # 匹配变量名
        \s                  # 匹配空格
        (?=in)              # 匹配 in
        ''',re.X | re.U)    # 设置编译选项，忽略模式中的注释
    vars.extend(r.findall(s))
    return vars
# 判断命令行是否有输入，没有则要求输入要处理的文件
if len(sys.argv) == 1:
    sour = input('请输入要处理的文件路径')
else:
    sour = sys.argv[1]
file = open(sour,encoding="utf-8")      # 打开文件
s = file.readlines()    # 将文件内容以行读入 s 中
file.close()            # 关闭文件
print('********************************')
print(sour,'中的函数有：')
print('********************************')
i = 0                   # i 为函数所在的行号
# 循环处理每一行，匹配其中的函数并输出函数所在的行号，以及函数的原型
for line in s:
    i = i + 1
    function = DealWithFunc(line)
    if len(function) == 1:
        print('Line: ',i,'\t',function[0])
print('********************************')
print(sour,'中的变量有：')
print('********************************')
i = 0                   # 此处 i 为变量所在的行号
# 循环处理每一行，匹配其中的变量，输出变量所在的行号，以及变量名
for line in s:
    i = i + 1
    var = DealWithVar(line)
    if len(var) == 1:
        print('Line: ',i,'\t',var[0])
```

运行脚本后，输入当前文件名 GetFunction.py，脚本将输出如下所示结果。

```
********************************
GetFunction.py 中的函数有：
********************************
Line:  6        DealWithFunc(s)
Line:  14       DealWithVar(s)
********************************
GetFunction.py 中的变量有：
********************************
Line:  7        r
Line:  15       vars
Line:  16       r
Line:  22       r
Line:  32       sour
Line:  34       sour
Line:  35       file
Line:  36       s
Line:  41       i
```

```
Line:  43      line
Line:  44      i
Line:  45      function
Line:  51      i
Line:  53      line
Line:  54      i
Line:  55      var
```

20.8 本章小结

正则表达式的功能非常强大，学习难度也较大，本章以尽量多的操作代码演示了正则表达式的用法。首先介绍了正则表达式的基本元字符和常用正则表达式分析。接着介绍了使用 Python 的 re 模块处理正则表达式，如用 match 函数进行搜索、使用 sub 函数进行内容替换、使用 split 函数分割等。接着介绍了将正则表达式编译为对象，以提供更高性能的方法。还介绍了正则表达式中的分组、匹配和搜索的结果对象、Match 对象的使用等内容。本章学习难度较大，读者应多编写代码进行学习、验证。掌握正则表达式的使用之后，在其他程序设计语言中也可以直接使用。

下一章将学习 Python 在科学计算领域的应用。

◆ ◆ ◆

第21章 科学计算

本章包括

- 认识和安装 NumPy
- 矩阵运算
- 使用 Matplotlib 绘制函数图形
- 认识和安装 SciPy
- 解线性方程组

科学计算是计算机应用的主要内容之一，开源软件 Scilab 和商业软件 MATLAB 都是以科学计算为主的应用软件，这些软件提供了数值分析、矩阵计算等科学计算的功能。在 Python 中安装第三方模块后，同样可以进行矩阵运算、数值分析等功能。

21.1 NumPy 和 SciPy 简介

NumPy 和 SciPy 是 Python 中用以实现科学计算的模块包。NumPy 主要提供了数组对象、基本的数组函数和傅里叶变换的相关函数。而 SciPy 依赖于 NumPy，在 SciPy 模块中，提供了更多的计算工具，还可绘制图形。

21.1.1 安装 NumPy 和 SciPy

NumPy 和 SciPy 都提供了 Windows 下的安装包。在 Windows 下安装 NumPy 和 SciPy 较为简单，只需根据所安装的 Python 版本选择相应的安装文件即可。

1. 安装 NumPy

以 Python 32 为例，在 Windows 下安装 NumPy 的步骤如下。

step 1 从 http://scipy.org/scipylib/download.html 下载 NumPy 在 Windows 下的安装包 numpy-1.7.1-win32-superpack-python3.2.exe。

step 2 双击运行安装程序，如图 21-1 所示。



图 21-1 NumPy 安装程序

step 3 单击【下一步】按钮，将出现如图 21-2 所示的【安装路径】界面，安装程序将自动找到安装 Python 3.2 的目录。

step 4 单击【下一步】按钮，进入【安装确认】界面，如图 21-3 所示。单击【下一步】按钮，开始进行安装，很快就可将所需要的文件安装完成。

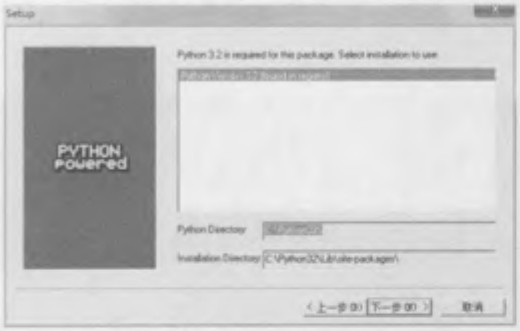


图 21-2 【安装路径】界面



图 21-3 【安装确认】界面

2. 安装 SciPy

同样以 Python 3.2 为例安装 SciPy，具体操作步骤如下。

step 1 从 SciPy 官方网站 http://scipy.org/scipylib/download.html 下载 SciPy 在 Windows 下的安装包 scipy-0.13.1-win32-superpack-python3.2.exe。

step 2 双击运行安装程序，如图 21-4 所示。

step 3 单击【下一步】按钮，将出现如图 21-5 所示的【安装路径】界面。



图 21-4 SciPy 安装程序



图 21-5 【安装路径】界面

step 4 单击【下一步】按钮，进入【安装确认】界面，如图 21-6 所示。单击【下一步】按钮，完成 SciPy 的安装。

需要注意的是，如果安装的 NumPy 比 SciPy 发布日期晚的话，则使用 SciPy 时将出现警告。



图 21-6 【安装确认】界面

21.1.2 NumPy 简介

NumPy 提供了 Python 没有提供的数组对象，使用 NumPy 的数组对象可以创建类似于 C 语言中的数组。

使用 NumPy 时，应首先导入 NumPy 模块，代码如下所示，在 Python 交互式命令行中，可以使用 NumPy 创建数组，并对数组进行简单的运算。

```
>>> import numpy                                          # 导入 NumPy 模块
>>> a = numpy.array((1,2,3,4,5))                          # 生成一个数组对象
>>> print(a)                                              # 打印数组 a
[1 2 3 4 5]
>>> b = numpy.array(([1,2,3],[4,5,6],[7,8,9]))           # 生成数组对象 b
>>> print(b)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> c = b + b                                             # 数组对象的加法运算
>>> print©
[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]
>>> d = c * 2                                             # 数组对象的乘法运算
>>> print(d)
[[ 4  8 12]
 [16 20 24]
 [28 32 36]]
>>> e = d / c                                             # 数组对象的除法运算
>>> print(e)
[[2 2 2]
 [2 2 2]
 [2 2 2]]
>>> print(b * e)                                          # 数组相乘
[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]
>>> numpy.sin(b)                                          # 求正弦
array([[ 0.84147098,  0.90929743,  0.14112001],
       [-0.7568025 , -0.95892427, -0.2794155 ],
       [ 0.6569866 ,  0.98935825,  0.41211849]])
>>> numpy.tan(b)                                          # 求正切
array([[ 1.55740772, -2.18503986, -0.14254654],
       [ 1.15782128, -3.38051501, -0.29100619],
       [ 0.87144798, -6.79971146, -0.45231566]])
>>> numpy.resize(b, [2,2])                                # 重新调整 b 大小，生成新数组
array([[1, 2],
       [3, 4]])
>>> numpy.resize(b, [3,4])                                # 重新调整 b 大小
array([[1, 2, 3, 4],
       [5, 6, 7, 8],
       [9, 1, 2, 3]])
>>> numpy.sum(b)                                          # 求 b 中所有元素的和
45
>>> zero = numpy.zeros((4,4))                             # 生成一个 4x4 的全为 0 的数组
```

```
>>> print(zero)
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
>>> one = numpy.ones((4,4))                    # 生成一个 4×4 的全为 1 的数组
>>> print(one)
[[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
```

21.1.3 SciPy 简介

SciPy 模块依赖于 NumPy，但是 SciPy 提供了更多的数学工具。使用 SciPy 不仅可以进行矩阵运算，还可以求解线性方程组、进行积分运算、优化等。SciPy 的功能非常接近 MATLAB。

下面所示的代码是在 Python 交互式命令行中使用 SciPy 模块。

```
>>> import scipy                               # 导入 scipy 模块
>>> a = scipy.mat('[1 2 3; 4 5 6; 7 8 9]')     # 生成一个矩阵
>>> a
matrix([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
>>> a * 2                                      # 矩阵乘以 2
matrix([[ 2,  4,  6],
        [ 8, 10, 12],
        [14, 16, 18]])
>>> 0.5 * a                                    # 0.5 乘以矩阵 a
matrix([[ 0.5,  1. ,  1.5],
        [ 2. ,  2.5,  3. ],
        [ 3.5,  4. ,  4.5]])
>>> b = scipy.mat('[1;2;3]')                   # 生成矩阵
>>> print(b)                                   # 输出矩阵
[[1]
 [2]
 [3]]
>>> a * b                                      # 矩阵相乘
matrix([[14],
        [32],
        [50]])
>>> scipy.sin(a)                               # 求正弦
matrix([[ 0.84147098,  0.90929743,  0.14112001],
        [-0.7568025 , -0.95892427, -0.2794155 ],
        [ 0.6569866 ,  0.98935825,  0.41211849]])
>>> scipy.tan(a)                               # 求正切
matrix([[ 1.55740772, -2.18503986, -0.14254654],
        [ 1.15782128, -3.38051501, -0.29100619],
        [ 0.87144798, -6.79971146, -0.45231566]])
>>> scipy.resize(a, (2,3))                     # 重新调整矩阵大小 2×3
array([[1, 2, 3],
       [4, 5, 6]])
>>> scipy.resize(a, (4,4))                     # 重新调整矩阵大小 4×4
array([[1, 2, 3, 4],
```

```
      [5, 6, 7, 8],
      [9, 1, 2, 3],
      [4, 5, 6, 7]])
>>> v = scipy.vander((1,6))                          # 生成 Vandermonde 矩阵
>>> v
array([[1, 1],
      [6, 1]])
>>> scipy.diff(v)                                    # 差分运算
array([[ 0],
      [-5]])
>>> from scipy import integrate                      # 导入 integrate 模块
>>> integrate.quad(lambda x: 2*x, 0, 6)              # 求积分 ∫_0^6 2xdx
(36.0, 3.9968028886505635e-013)
>>> integrate.quad(lambda x: 1/(1 + x ** 2), 0, 1)   # 求积分 ∫_0^1 dx/(1+x^2)
(0.78539816339744839, 8.7196712450215814e-015)
```

21.2 矩阵运算和解线性方程组

使用 SciPy 可以完成矩阵分解、线性方程组求解和多项式求根等数学运算。SciPy 中提供的函数名与 MATLAB 中的函数名大部分都相同，用法也差不多。熟悉 MATLAB 的用户可以很快地熟悉 SciPy。

21.2.1 矩阵运算

除了基本的矩阵乘除运算外，还可以使用 SciPy 做矩阵分解运算、求逆运算等。SciPy 中的 linalg 模块提供了和线性代数相关的函数，可以用于对矩阵进行运算。linalg 模块中的函数如表 21-1 所示。

表 21-1 linalg 模块中的常用函数

函数	描述
inv	求解矩阵的逆
det	求解方阵的行列式
norm	求解向量的模
lstsq	求最小二乘法解
eig	求解特征值和特征向量
orth	求解矩阵的标准正交基
lu	矩阵的 LU 分解
qr	矩阵的 QR 分解
cholesky	矩阵的 Cholesky 分解

下面的代码是在 Python 的交互式命令行中使用 SciPy 中的 linalg 模块对矩阵进行运算。

```
>>> import scipy                                     # 导入 SciPy 模块
>>> from scipy import linalg                         # 导入 linalg 模块
>>> a = scipy.mat('[1 2 3; 2 2 1; 3 4 3]')          # 生成矩阵
```

```
>>> b = linalg.inv(a)                                    # 求 a 的逆矩阵
>>> print(b)
[[ 1.   3.  -2. ]
 [-1.5 -3.   2.5]
 [ 1.   1.  -1. ]]
>>> a * b                                                # 求 a×b，有误差，应为对角矩阵
matrix([[  1.00000000e+00,  -4.44089210e-16,  -4.44089210e-16],
        [  0.00000000e+00,   1.00000000e+00,  -2.22044605e-16],
        [ -4.44089210e-16,   0.00000000e+00,   1.00000000e+00]])
>>> linalg.det(a)                                        # 求 a 的行列式值，有误差，应为 2
1.9999999999999996
>>> a = scipy.mat('[1 2 3]')
>>> linalg.norm(a)                                       # 求 a 的模
3.74165738677
```

#如下所示步骤求超定方程 $\begin{cases} x_1 - x_2 = 1 \\ -x_1 + x_2 = 2 \\ 2x_1 - 2x_2 = 3 \\ -3x_1 + x_2 = 4 \end{cases}$ 的最小二乘解

```
>>> a = scipy.mat('[1 -1; -1 1; 2 -2; -3 1]')            # 建立系数矩阵
>>> a                                                    # 检验系数矩阵
matrix([[ 1, -1],
        [-1,  1],
        [ 2, -2],
        [-3,  1]])
>>> b = scipy.mat('[1;2;3;4]')                           # 建立常数项向量
>>> b
matrix([[1],
        [2],
        [3],
        [4]])
>>> x,y,z,w = linalg.lstsq(a,b)                          # 求最小二乘解
>>> x                                                    # x 为解
array([[-2.41666667],
       [-3.25      ]])
>>> y
array([ 9.83333333])
>>> z
2
>>> w
array([ 4.56605495,  1.07291295])
>>> a = scipy.mat('[-1 1 0; -4 3 0; 1 0 2]')
>>> print(a)
[[-1  1  0]
 [-4  3  0]
 [ 1  0  2]]
>>> x,y = linalg.eig(a)                                  # 求 a 的特征值
>>> x
array([ 2.        +0.j,  0.99999998+0.j,  1.00000002+0.j])  # a 的特征值，有误差，应为[2, 1, 1]
>>> y                                                    # a 的特征向量
array([[ 0.        ,  0.40824829, -0.40824829],
       [ 0.        ,  0.81649658, -0.81649658],
```

```
      [ 1.        , -0.40824829,  0.40824829]])
>>> a = scipy.mat('[1 2 3; 0 1 2; 2 4 1]')
>>> print(a)
[[1 2 3]
 [0 1 2]
 [2 4 1]]
>>> x,y,z = linalg.lu(a)                                    # 求 a 的 LU 分解
>>> x
array([[ 0.,  0.,  1.],
       [ 0.,  1.,  0.],
       [ 1.,  0.,  0.]])
>>> y                                                       # L 矩阵
array([[ 1. ,  0. ,  0. ],
       [ 0. ,  1. ,  0. ],
       [ 0.5,  0. ,  1. ]])
>>> z                                                       # U 矩阵
array([[ 2. ,  4. ,  1. ],
       [ 0. ,  1. ,  2. ],
       [ 0. ,  0. ,  2.5]])
>>> a = scipy.mat('[16 4 8;4 5 -4; 8 -4 22]')
>>> print(a)
[[16  4  8]
 [ 4  5 -4]
 [ 8 -4 22]]
>>> linalg.cholesky(a)                                      # 求 a 的 Cholesky 分解
array([[ 4.,  1.,  2.],
       [ 0.,  2., -3.],
       [ 0.,  0.,  3.]])
```

21.2.2 解线性方程组

SciPy 中的 linalg 模块提供了基本的解线性方程组的函数，只要给出方程的系数矩阵和常数项向量，就可以获得方程组的解。对于非线性方程组，可以采用数值解法。

linalg 模块中的 solve 函数可以用于解线程方程组，例如，下面的方程组。

$$\begin{cases} 2x_1 + x_2 - 5x_3 + x_4 = 8 \\ x_1 - 3x_2 - 6x_4 = 9 \\ 2x_2 - x_3 + 2x_4 = -5 \\ x_1 + 4x_2 - 7x_3 + 6x_4 = 0 \end{cases}$$

其系数矩阵为：$\begin{bmatrix} 2 & 1 & -5 & 1 \\ 1 & -3 & 0 & -6 \\ 0 & 2 & -1 & 2 \\ 1 & 4 & -7 & 6 \end{bmatrix}$；其常数项向量为：$\begin{bmatrix} 8 \\ 9 \\ -5 \\ 0 \end{bmatrix}$。

要对这个线性方程进行求解，则可在 Python 交互式命令行中依次输入以下命令来进行。

```
>>> import scipy
>>> from scipy import linalg
>>> a = scipy.mat('[2 1 -5 1;1 -3 0 -6;0 2 -1 2;1 4 -7 6]')
>>> print(a)
[[ 2  1 -5  1]
```

```
 [ 1 -3  0 -6]
 [ 0  2 -1  2]
 [ 1  4 -7  6]]
>>> b = scipy.mat('[8;9;-5;0]')
>>> print(b)
[[ 8]
 [ 9]
 [-5]
 [ 0]]
>>> linalg.solve(a,b)
array([[ 3.],
       [-4.],
       [-1.],
       [ 1.]])
```

21.3 使用 Matplotlib 绘制函数图形

在 Python 中，可以使用 Matplotlib 模块来绘制二维图形，Matplotlib 模块依赖于 NumPy 模块和 tkinter 模块（一个模块提供数值计算支持，一个模块提供图形界面支持）。Matplotlib 可以绘制多种形式的图形，包括普通的线图、直方图、饼形图、散点图以及误差线图等。

Matplotlib 中的大部分函数都和 MATLAB 中的函数名相同，熟悉 MATLAB 的用户可以很快地掌握 Matplotlib。

21.3.1 安装 Matplotlib

要使用 Matplotlib 模块进行绘制，首先必须下载安装相关的模块。除了需要安装 Matplotlib 模块外，建议再安装一个 IPython，这样，就能像 MATLAB 一样在交互式命令下绘制图形了。

1. 下载安装 Matplotlib

Matplotlib 提供了 Windows 下的安装程序，以 Python 3.2 为例，其安装步骤如下。

step 1 从网站 http://sourceforge.net/projects/matplotlib/files/下载 matplotlib-1.3.1.win32-py3.2.exe 安装程序。

step 2 双击运行安装程序，如图 21-7 所示。

step 3 单击【下一步】按钮，进入【选择安装路径】界面，如图 21-8 所示。



图 21-7 Matplotlib 安装程序

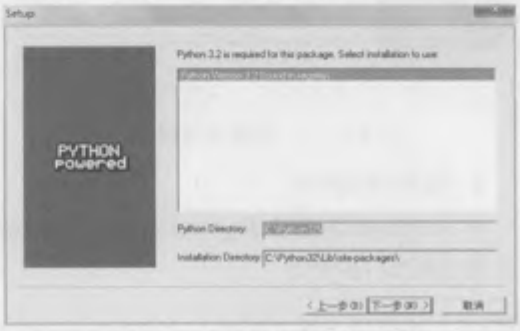


图 21-8 【选择安装路径】界面

step 4 单击【下一步】按钮，进入【安装确认】界面，如图 21-9 所示。单击【下一步】按钮，

完成 Matplotlib 的安装。

2. 下载安装 IPython

为了能像 MATLAB 一样在交互式命令下绘制图形，需要安装 IPython。IPython 是一个 Python 的交互式命令窗口，比默认的 Python 命令窗口要好用得多。例如，IPython 支持变量自动补全、自动缩进等，还内置了许多很有用的功能和函数。IPython 提供了对 Matplotlib 的支持。以 Python 3.2 为例，在 Windows 下安装 IPython 步骤如下。

step 1 从网站 https://github.com/ipython/ipython/releases 下载 Windows 下的安装程序 ipython-1.1.0.py3-win32.exe。

step 2 双击运行安装程序，如图 21-10 所示。



图 21-9 【安装确认】界面



图 21-10 IPython 安装程序

step 3 单击【下一步】按钮，IPython 将搜索本机所安装的 Python 版本，根据需要选择要使用的 Python 版本，如图 21-11 所示。

step 4 单击【下一步】按钮，进入【安装确认】界面，如图 21-12 所示。单击【下一步】按钮，完成 IPython 的安装。



图 21-11 选择安装路径



图 21-12 【安装确认】界面

3. 安装 readline

在 Windows 环境下，IPython 还需要安装 readline 来模拟 UNIX/Linux 下的 readline 功能。readline 的安装过程如下所示。

step 1 从网站 https://pypi.python.org/pypi/pyreadline/2.0 下载 readline 安装程序 pyreadline-2.0.win32.exe。

step 2 双击运行安装程序，如图 21-13 所示。

step 3 单击【下一步】按钮，readline 将搜索本机所安装的 Python 版本，并根据需要选择要使用的 Python 版本，如图 21-14 所示。



图 21-13 readline 安装程序



图 21-14 【安装路径】界面

step 4 单击【下一步】按钮，进入【安装确认】界面，如图 21-15 所示。单击【下一步】按钮，完成 readline 的安装。



图 21-15 【安装确认】界面

step 5 依次单击【开始】|【所有程序】|【IPython (Py3.2 32 bit)】|【IPython (pylab mode)】命令，将打开如图 21-16 所示的 IPython 窗口，在其中输入 Python 命令即可执行。



图 21-16 IPython 窗口

21.3.2 使用 Matplotlib 绘制图形

使用 Matplotlib 绘制图形的操作主要是使用 plot 函数进行，plot 函数和 MATLAB 中的 plot 函

数用法相同。Matplotlib 还包含了一些用于设置 X，Y 轴标签文本以及图形标题的函数，如表 21-2 所示。

表 21-2 Matplotlib 中设置文字的函数

函数	描述
xlabel	设置 X 轴标签
ylabel	设置 Y 轴标签
title	设置绘图标题
text	在指定坐标处输出文字
figtext	在绘制的图形上添加文字

另外，Matplotlib 还支持一部分 Tex 排版命令，可以较好地显示数学公式。下面所示的代码是在 IPython 交互式命令行中使用 Matplotlib 绘制图形（斜体为用户输入部分）。

```
# 绘制正弦曲线，如图 21-17 所示。
In [1]: from pylab import *
In [2]: t = arange(0.0, 2.0, 0.05)
In [3]: s = sin(2*pi*t)
In [4]: plot(t,s)
Out[4]: [<matplotlib.lines.Line2D instance at 0x0181EBC0>]
# 绘制余弦曲线，如图 21-18 所示。
In [5]: figure(2)                                          # 新建一个绘图窗口
Out[5]: <matplotlib.figure.Figure instance at 0x019D06E8>
In [6]: s = cos(2*pi*t)
In [7]: plot(t,s)
Out[7]: [<matplotlib.lines.Line2D instance at 0x019DB4E0>]
# 同时绘制正弦曲线和余弦曲线，如图 21-19 所示。
In [8]: clf()                                              # 清除图形
In [9]: plot(t,s,linestyle='-',marker='o')
Out[9]: [<matplotlib.lines.Line2D instance at 0x019EA440>]
In [10]: s = sin(2*pi*t)
In [11]: plot(t,s,linestyle='-.',marker='+')
Out[11]: [<matplotlib.lines.Line2D instance at 0x019F0D00>]
In [12]: xlabel('X')
Out[12]: <matplotlib.text.Text instance at 0x019DFEB8>
In [13]: ylabel('Y')
Out[13]: <matplotlib.text.Text instance at 0x019E57D8>
In [14]: title('sin(x) and sin(y)')
Out[14]: <matplotlib.text.Text instance at 0x019EA2B0>
# 绘制函数 y = 10/(1+x^2) 曲线，如图 21-20 所示。
In [15]: clf()
In [16]: x = arange(-5, 5, 0.1)
In [17]: y = 10/(1+x**2)
In [18]: plot(x,y)
Out[18]: [<matplotlib.lines.Line2D instance at 0x019FB468>]
In [19]: xlabel('X')
Out[19]: <matplotlib.text.Text instance at 0x019F4080>
In [20]: ylabel('Y')
Out[20]: <matplotlib.text.Text instance at 0x019F4800>
```

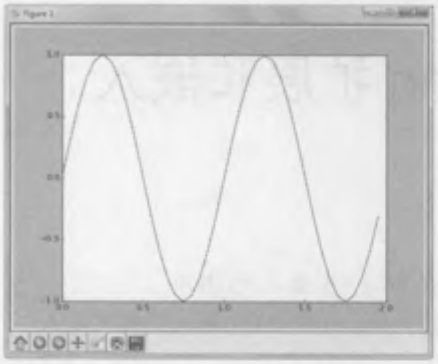


图 21-17 正弦曲线

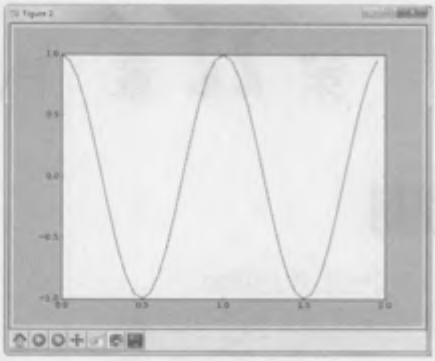


图 21-18 余弦曲线

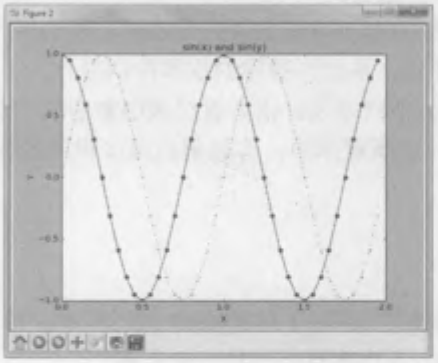


图 21-19 同时绘制正弦曲线和余弦曲线

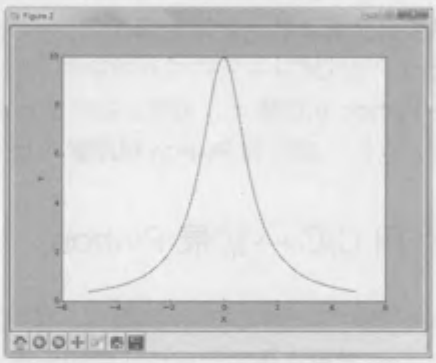


图 21-20 函数 $y=\frac{10}{1+x^2}$ 曲线



如果在导入 pylab 模块时提示需要 dateutil 模块，则可到 https://pypi.python.org/pypi/python-dateutil 中下载 python-dateutil-2.2.win32-py3.2.exe 安装程序，并进行安装。另外，可能还需要安装 pyparsing 和 six 模块，这两个模块可分别到以下网址下载。

在 https://pypi.python.org/pypi/pyparsing/2.0.1 中下载 pyparsing-2.0.1.win32-py3.2.exe，并安装。

在 https://pypi.python.org/pypi/six/1.4.1 中下载 six-1.4.1.tar.gz 安装包，解压后执行以下命令进行安装：

```
C:\six-1.4.1>Python setup.py install
```

21.4 本章小结

Python 通过第三方模块的支持，可方便地进行矩阵运算、数值分析等操作。本章介绍了完成这些功能的两个模块：NumPy 模块和 SciPy 模块。其中，NumPy 模块是基础，SciPy 模块是在 NumPy 模块的基础上运行的。本章介绍了这两个模块的下载和安装，然后介绍了使用这两个模块进行矩阵运算、解线性方程组的方法。本章最后还介绍了使用 Matplotlib 模块绘制分析图形的操作。

下一章将进入另一个主题：用 C/C++扩展 Python。

◆ ◆ ◆

第 22 章 Python 扩展和嵌入

本章包括

- 用 C/C++扩展 Python
- 通过 SWIG 编写 Python 扩展
- 在 C/C++中嵌入 Python
- 使用 Boost.Python 简化扩展和嵌入

由于 Python 是解释型的脚本语言，因此程序员编写程序效率极高，但程序的执行速度较慢。在某些需要提高脚本执行效率的情况下，可以考虑使用如 C/C++这类执行效率更高的程序语言来扩展 Python，用 C/C++来完成对效率要求较高的部分，从而使项目的效率得到提升。

由于 Python 功能强大，用很少的代码就能完成其他程序设计语言看起来很复杂的功能。为了简化程序，这时，就可将 Python 程序嵌入到 C/C++这类程序中，可显著地减少程序的代码量。

22.1 用 C/C++扩展 Python

Python 提供了支持 C/C++的接口，可以方便地使用 C/C++对 Python 进行扩展。用 C/C++编写的 Python 扩展主要用于完成底层的系统操作，以及提高程序的执行速度等。

22.1.1 VS2008 编译环境的设置

Python 提供了接口 API，通过使用 API 函数可以编写 Python 扩展。在 Windows 下可以使用 VS2008 来编译 Python 扩展。在 UNIX 和 Linux 下则可使用 gcc 来编译。

1. 设置编程环境

使用 VS2008 时，需要设置一些头文件以及库文件的包含目录，设置过程如下。

step 1 启动 VS2008，单击【工具】|【选项】命令，打开如图 22-1 所示的对话框。



图 22-1 【选项】对话框

step 2 双击左侧列表树中的【项目和解决方案】项，选择【VC++目录】项，如图 22-2 所示。



图 22-2 选择【VC++目录】项

step 3 选择【显示以下内容的目录】下拉列表框中的【包含文件】项，将 Python 安装目录下的 include 目录添加到列表中，如图 22-3 所示。



图 22-3 添加头文件

step 4 选择【显示以下内容的目录】下拉列表框中的【库文件】项，将 Python 安装目录下的 libs 目录添加到列表中，如图 22-4 所示。



图 22-4 添加库文件

step 5 单击【确定】按钮，完成操作。

2. 创建工程

在 VS2008 中创建 Python 扩展的过程如下。

step 1 单击菜单【文件】|【新建】|【项目】命令，弹出【新建项目】对话框。在左侧【项目类型】列表中单击选择【Visual C++】下的【Win32】项，在右侧选择【Win32 项目】，在对话框下方的【名称】文本框中输入项目名"MyExt"，并选择保存项目的位置，如图 22-5 所示。

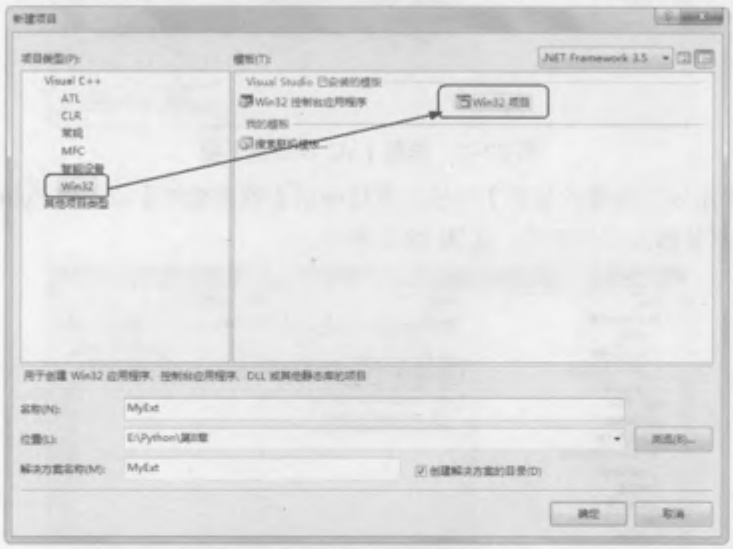


图 22-5 【新建项目】对话框

step 2 单击【确定】按钮，弹出如图 22-6 所示的【Win32 应用程序向导】对话框，单击【下一步】按钮，显示如图 22-7 所示界面，单击【DLL】单选按钮。



图 22-6 【Win32 应用程序向导】对话框



图 22-7 单击【DLL】单选按钮

step 3 单击【完成】按钮，完成项目新建，在 VS2008 窗口左侧的【解决方案资源管理器】对话框中可以看到新建的文件，如图 22-8 所示。



图 22-8 【解决方案资源管理器】对话框

step 4 在【解决方案资源管理器】对话框中只保留 MyExt.cpp 文件，将【头文件】和【源文件】中的其他文件删除。

step 5 接下来就可以开始编写代码了。在【解决方案资源管理器】对话框中双击打开“MyExt.cpp”文件，删除里面的内容，然后输入以下代码。

```
#include <python.h>
#include <windows.h>

static PyObject * show(PyObject *self, PyObject *args)
{
    char *message;
    const char *title = NULL;
    HWND hwnd = NULL;
    int r;
    if (!PyArg_ParseTuple(args, "iss", &hwnd, &message, &title))
        return NULL;
    r = MessageBox(hwnd,message, title, MB_OK);
    return Py_BuildValue("i", r);
}

static PyMethodDef myextMethods[] =
{
    {"show", show, METH_VARARGS,"show a messagebox"},
    {NULL, NULL, 0, NULL}
};

static struct PyModuleDef myextmodule = {
   PyModuleDef_HEAD_INIT,
   "myext",
   NULL,
   -1,
   myextMethods
};

PyMODINIT_FUNC PyInit_myext()
{
    PyObject *pModule;
    pModule = PyModule_Create(&myextmodule);
   return pModule;
}
```

step 6 将扩展功能的代码编写完成后，还需要对编译链接进行设置，这样才能正确地编译链接生成 Python 的扩展文件。在【解决方案资源管理器】中，用鼠标右击项目名称“MyExt”，从弹出的快捷菜单中选择【属性】命令，打开如图 22-9 所示的【MyExt 属性页】对话框。

step 7 单击左上方的【配置】下拉列表，从中选择【Release】。接着在左侧列表中依次选择【配置属性】|【C/C++】|【预编译头】，然后在右侧【创建/使用预编译头】中选择【不使用预编译头】项。

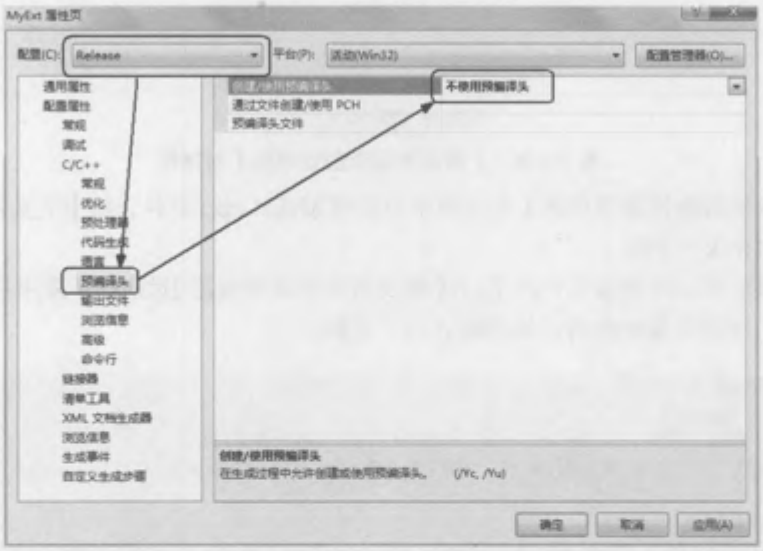


图 22-9 【MyExt 属性页】对话框

step 8 在【MyExt 属性页】对话框左侧依次选择【配置属性】|【链接器】|【常规】，接着在右侧【输出文件】中输入文件名“.\Release\myext.pyd”（注意文件名的大小写），如图 22-10 所示。



图 22-10 设置输出文件

step 9 在【MyExt 属性页】对话框左侧依次选择【配置属性】|【常规】，接着在右侧【字符集】下拉列表中选择【未设置】，如图 22-11 所示。若不进行这步设置，则编译程序时可能会出现下面所示的提示。

```
error C2664: "MessageBoxW": 不能将参数 2 从"char *"转换为"LPCWSTR"
```

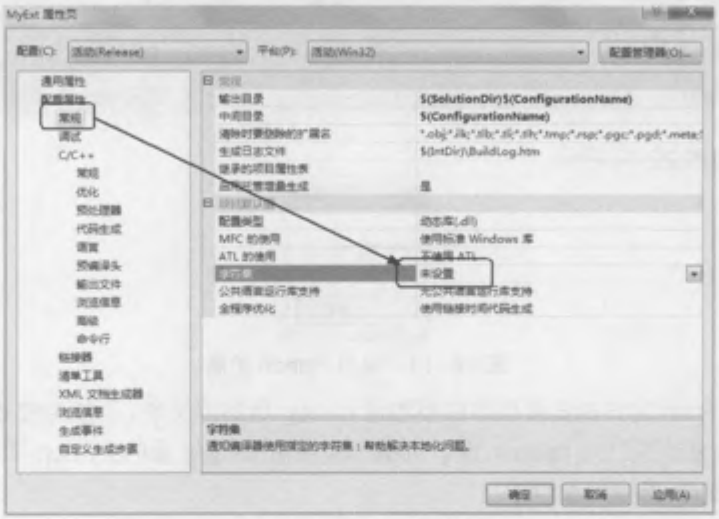


图 22-11 设置字符集

step 10 设置完成后，单击【确定】按钮，完成项目的设置。

step 11 单击菜单【生成】|【批生成】命令，弹出如图 22-12 所示对话框，勾选 Release 配置行右侧的【生成】复选框。

注意，以上都只是进行了 Release 配置的设置，如果要编译 Debug 配置，则需要进行类似的设置。然后就可在图 22-12 所示对话框中勾选 Debug 配置行的【生成】复选框了。



图 22-12 【批生成】对话框

step 12 在图 22-12 对话框中，单击右上角的【生成】按钮，即可将前面编写的扩展程序进行编译、链接，最后在工程目录下的 Release 目录生成“myext.pyd”文件。这就是自己编写的 Python 扩展。

编译生成 pyd 扩展之后，接下来就可以编写 Python 程序来使用扩展中提供的功能。下面编写名为“usemyext.py”的程序，调用“myext.pyd”扩展中的 show 函数显示一个对话框。具体代码如下。

```
# -*- coding:utf-8 -*-
# file: usemyext.py
#
import myext                                          # 导入 myext 模块
print(myext.show(0, 'Extend Python', 'Python'))       # 调用 show 函数
```

脚本运行后如图 22-13 所示。



图 22-13 使用 Python 扩展

注意，由于 Python 官方的安装程序中不包含 debug 版的库文件，不能生成 debug 版的 Python 扩展，因此上述设置都是针对 release 版。如果需要生成 debug 版的 Python 扩展，则需要自己编译 Python 生成 debug 版的库文件。

22.1.2 Python 扩展程序的结构

一般的 Python 扩展程序中应包含以下三部分内容。

1. 初始化函数

初始化函数是必须的，用于 Python 解释器对模块进行正确的初始化。初始化函数的函数名必须以 PyInit_开头，并加上模块的名字。例如，21.1.1 节中初始化函数的函数名为“PyInit_myext”，其中“myext”为模块名。函数“PyInit_myext”代码如下。

```
PyMODINIT_FUNC PyInit_myext()
{
    PyObject *pModule;
    pModule = PyModule_Create(&myextmodule);
    return pModule;
}
```

其中，PyMODINIT_FUNC 为 Python.h 头文件中定义的宏，在 Windows 下其相当于 __declspec(dllexport) void，即将 PyInit_myext 声明为 void 型，并且将其设为 DLL 文件的导出函数。PyModule_Create 是在 modsupport.h 头文件中定义的一个宏，用来生成一个 Python 扩展。

2. 方法列表

方法列表中包含了 Python 扩展中的所有可以调用的函数方法。方法列表应该被声明为“static PyMethodDef”。22.1.1 节实例中的方法列表如下。

```
static PyMethodDef myextMethods[] =
{
    {"show", show, METH_VARARGS,"show a messagebox"},
```

```
    {NULL,NULL , 0, NULL }
};
```

每一个函数方法对应着方法列表中的由大括号包围的一项。大括号中由四部分组成，模块中的方法名（即在 Python 调用的方法名称），与之对应的 Python 扩展中的函数名、函数调用方法及方法描述。其中，函数调用方法应该为“METH_VARARGS”或者“METH_VARARGS | METH_KEYWORDS”。也可以将函数调用方法设置为 0。方法列表中的最后一项应该由多个 NULL 组成的项来表示结束。

3. 函数实现

方法列表中包含了模块中方法对应的 C 语言函数实现。在 Python 扩展中，所有的函数都应该被声明为“PyObject *”型，每个函数都应当含有两个“PyObject *”型的参数。在 22.1.1 节的实例中，模块方法的实现函数如下。

```
PyObject *show(PyObject *self, PyObject *args)
{
    char *message;
    const char *title = NULL;
    HWND hwnd = NULL;
    int r;
    if (!PyArg_ParseTuple(args, "iss", &hwnd, &message, &title))
        return NULL;
    r = MessageBox(hwnd,message, title, MB_OK);
    return Py_BuildValue("i", r);
}
```

其中，参数 self 只有在函数为 Python 的内置方法时才被使用，其余情况下，self 为一个空指针。参数 args 为在 Python 中向方法传递的参数。如果在方法列表中，指定的函数调用方法为“METH_VARARGS”，则在函数中使用 PyArg_ParseTuple 处理参数。如果在方法列表中指定的函数调用方法为“METH_VARARGS | METH_KEYWORDS”，则应该使用 PyArg_ParseTupleAndKeywords 处理参数。其中 PyArg_ParseTuple 的函数原型如下。

```
int PyArg_ParseTuple( PyObject *args, const char *format, ...)
```

其参数含义如下。

- args　　传递的参数。
- format　　参数类型描述。

PyArg_ParseTuple 为可变参数函数，其后的参数即在函数中接收 Python 中传递参数的变量。在上述 show 函数中，要使用 3 个参数，分别为 hwnd、message 和 title。用 PyArg_ParseTuple 将其作为参数，使用“&”向 hwnd、message 和 title 传递值，即将 Python 向 show 方法传递的参数依次赋值给 hwnd、message 和 title。

PyArg_ParseTuple 函数中的 format 参数指定了其后参数的类型，在 show 函数中，format 参数为“iss”则表示 hwnd 为整型、message 和 title 为字符串。常见的指定参数类型的字符如表 22-1 所示。

表 22-1 格式化字符

格式化字符	C 数据类型	Python 类型
s	char*	字符串
s#	char*, int	字符串及长度
z	char*	与 s 相同，但可以为 NULL
z#	char*, int	与 s#相同，但可以为 NULL
i	int	整型
l	long int	整型
c	char	单个字符的字符串
f	float	浮点型
d	double	浮点型

22.1.3 在 Python 扩展中使用 MFC

在 Windows 下，使用 MFC 可以方便地进行 GUI 编程。MFC 对基本的 SDK API 函数进行了封装，比直接使用 SDK API 更为简便。

PythonWin 中提供了部分 MFC 中的函数，因此，可方便地在 Python 中调用这些 MFC 函数进行 GUI 编程。在 Python 扩展中，使用 MFC 与 22.1.2 节中的例子略有不同。下面给出一个在 Python 扩展中使用 MFC 创建一个对话框的例子。整个过程如下。

step 1 启动 VS2008，单击菜单【文件】|【新建】|【项目】命令，弹出【新建项目】对话框。选择左侧列表中的【MFC】项，接着在右侧选择【MFC DLL】，在下方【名称】文本框中输入项目名“UseMFC”，并选择保存项目的位置，如图 22-14 所示。



图 22-14 新建 MFC DLL 项目

step 2 单击【确定】按钮，弹出如图 22-15 所示的【MFC DLL 向导】对话框。单击【下一步】按钮，显示如图 22-16 所示界面，选择【使用共享 MFL DLL 的规则 DLL】动态链接方式。

该方式需要 MFC DLL 的支持，如果选择【带静态链接 MFC 的规则 DLL】单选按钮，则使用静态链接的方式，这样会增大生成的 Python 扩展的体积。



图 22-15 【MFC DLL 向导】对话框



图 22-16 【应用程序设置】界面

step 3 单击【完成】按钮，完成项目的创建。

step 4 如图 22-17 所示，用鼠标右击项目名称【UserMFC】，从弹出的快捷菜单中选择【添加】|【资源】命令，将弹出如图 22-18 所示的【添加资源】对话框。

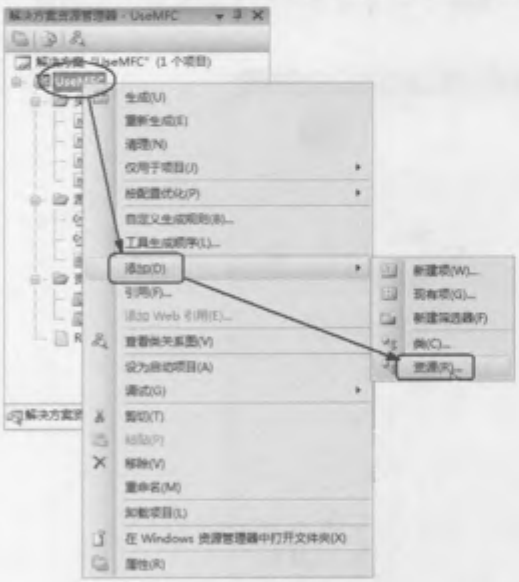


图 22-17 快捷菜单



图 22-18 【添加资源】对话框

step 5 单击选择【Dialog】，再单击【新建】按钮，在项目中新建一个对话框。向对话框中添加 Edit 控件和 Static Text 控件，将其修改为如图 22-19 所示的形式。

step 6 在创建的对话框上单击右键，在弹出的快捷菜单中选择【添加类】命令，弹出如图 22-20 所示的【添加类】对话框。

step 7 在图 22-20 所示对话框中输入类名为“CInput”，其余按照默认选项。单击【完成】按钮，将对话框创建为一个类。

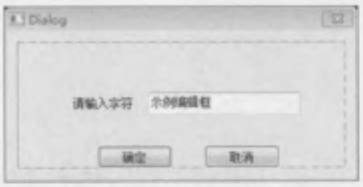


图 22-19　创建对话框



图 22-20　为对话框添加类

step 8　在创建的对话框中的文本框上单击右键，从弹出的快捷菜单中选择【添加变量】命令，将弹出如图 22-21 所示对话框，在【变量名】文本框中输入“m_input”，在【控件 ID】中选择“IDC_EDIT1”（就是图 22-19 中添加的文本框）为其添加变量，用来获取文本框中输入的字符串。在【控件 ID】中选择“Value”，在【变量类型】中选择“CString”，单击【完成】按钮，完成添加变量。



图 22-21　【添加成员变量向导】对话框

step 9　打开 UseMFC.cpp 文件，将以下头文件添加到其中。

```
#include "Input.h"
#include <Python.h>
```

然后将如下所示代码添加到 UseMFC.cpp 文件中。

```
PyObject *show(PyObject *self, PyObject *args)
{
```

```
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    CInput dia;
    dia.DoModal();
    return Py_BuildValue("s", dia.m_input);
}
static PyMethodDef UseMFCMethods[] =
{
    {"show", show, METH_VARARGS,"show a messagebox"},
    {NULL,NULL,0,NULL}
};
static struct PyModuleDef UseMFCmodule = {
   PyModuleDef_HEAD_INIT,
   "UseMFC",
   NULL,
   -1,
   UseMFCMethods
};
PyMODINIT_FUNC PyInit_UseMFC()
{
    PyObject *pModule;
    pModule = PyModule_Create(&UseMFCmodule);
   return pModule;
}
```

step 10 打开 UseMFC.def 文件，将初始化函数添加到 UseMFC.def 文件中。def 文件用来告诉连接器 DLL 文件的导出函数，相当于使用 PyMODINIT_FUNC 声明初始化函数。UseMFC.def 文件内容如下所示。

```
; UseMFC.def : Declares the module parameters for the DLL.

LIBRARY      "UseMFC"
DESCRIPTION  'UseMFC Windows Dynamic Link Library'

EXPORTS
   ; Explicit exports can go here
    PyInit_UseMFC
```

step 11 按照 22.1.1 节中创建工程的第（6）～（11）步操作，完成 Python 扩展的编译、链接操作。在项目目录的 Release 子目录中得到 UseMFC.pyd 扩展库文件。

step 12 在项目目录的 Release 子目录中编写如下所示的 UseMFC.py 脚本，调用编译好的 UseMFC 模块。

```
# -*- coding:utf-8 -*-
# file: UseMFC.py
#
import UseMFC                          # 导入 UseMFC 模块
input = UseMFC.show()                      # 调用 show 函数
print('刚才输入的是: ',input)
```

step 13 运行脚本后，在文本框中输入 “Hi,Python and MFC!”，如图 22-22 所示。单击【确定】按钮后，在命令窗口中将输出如图 22-23 所示的内容。



图 22-22 脚本运行后弹出的对话框



图 22-23 获得文本框中的文本

22.2 在 C/C++中嵌入 Python

在 C/C++中嵌入 Python，可以使用 Python 提供的强大功能，提高 C/C++的编程效率。通过在 C/C++程序中嵌入 Python，可以替代动态链接库形式的接口。这样可以方便地根据需要修改脚本代码，而不用重新编译链接由 C/C++生成的二进制动态链接库。

22.2.1 高层次的嵌入 Python

使用 PythonPython/C API 可以在较高层次上嵌入 Python。所谓的高层次嵌入，主要是指程序与脚本间没有交互。例如，在 VS2008 中新建一个空“Win32 控制台应用程序”，在工程中新建一个 C 源文件。将以下代码添加到其中。

```
#include <Python.h>
int main()
{
    Py_Initialize();                                /* Python 解释器初始化 */
    PyRun_SimpleString("print('hi,python!')"); /* 运行字符串 */
    Py_Finalize();                                  /* 结束 Python 解释器，释放资源 */
    return 0;
}
```

编译工程，运行程序后输出如下所示。

```
hi,python!
```

可以看到程序很简单，只使用了三个函数。其中，Py_Initialize 函数原型如下。

```
void Py_Initialize()
```

在嵌入 Python 脚本中必须使用该函数，它初始化 Python 解释器。在使用其他的 Python/C API 之前，必须先调用 Py_Initialize 函数。其中，PyRun_SimpleString 函数用来执行一段 Python 代码。其函数原型如下。

```
int PyRun_SimpleString(const char *command)
```

在程序的最后使用了 Py_Finalize 函数，其原型如下。

```
void Py_Finalize()
```

Py_Finalize 函数用于关闭 Python 解释器，释放解释器所占用的资源。

除了使用 PyRun_SimpleString 函数外，还可以使用 PyRun_SimpleFile()函数来运行 ".py" 脚本文件。其原型如下。

```
int PyRun_SimpleFile( FILE *fp, const char *filename)
```

其参数含义如下。

- fp　　打开的文件指针。
- filename　要运行的 Python 脚本文件名。

在 Windows 下使用该函数时，需要注意所使用的编译器版本。由于官方发布的 Pyhon 3.2.5 是由 Visual Studio 2008 编译的，如果使用其他版本的编译器，则由于版本差异将导致 FILE 的定义有区别，因此使用其他版本的编译器会导致程序崩溃。

为了简便起见，可以使用如下方式来代替 PyRun_SimpleFile 函数实现同样的功能。

```
PyRun_SimpleString("execfile('file.py')");      # 使用 execfile 运行 Python 脚本文件
```

22.2.2 较低层次嵌入 Python

在 22.2.2 节的例子中，只需要使用几个简单的函数就完成了在 C 语言中嵌入 Python 脚本的操作。但如果需要在 C 程序中向 Python 脚本传递参数，或者获得 Python 脚本的返回值，则要使用更多的函数来编写 C 程序。

由于 Python 有自己的数据类型，因此在 C 程序中要使用专门的 API 对相应的数据类型进行转换。常用的函数有以下几种。

1. 数字与字符串处理

在 Python/C API 中，提供了 Py_BuildValue()函数对数字和字符串进行转换处理，使之变成 Python 中相应的数据类型。其函数原型如下。

```
PyObject* Py_BuildValue( const char *format, ...)
```

其参数含义如下。

- format　格式化字符串，如表 22-1 所示。

Py_BuildValue()函数中剩余的参数，即要转换的 C 语言中的整型、浮点型或者字符串等，其返回值为 PyObject 型的指针。在 C 中，所有的 Python 类型都被声明为 PyObject 型。

2. 列表操作

在 Python/C API 中，提供了 PyList_New()函数用以创建一个新的 Python 列表。PyList_New()函数的返回值为所创建的列表。其函数原型如下。

```
PyObject* PyList_New( Py_ssize_t len)
```

其参数含义如下。

- len　　所创建列表的长度。

当列表创建以后，可以使用 PyList_SetItem()函数向列表中添加项。其函数原型如下。

```
int PyList_SetItem( PyObject *list, Py_ssize_t index, PyObject *item)
```

其参数含义如下。

- list　　要添加项的列表。
- index　　所添加项的位置索引。
- item　　所添加项的值。

同样可以使用Python/C API中的PyList_GetItem()函数来获取列表中某项的值。PyList_GetItem()函数返回项的值。其函数原型如下。

```
PyObject* PyList_GetItem( PyObject *list, Py_ssize_t index)
```

其参数含义如下。

- list　　要进行操作的列表。
- index　　项的位置索引。

Python/C API 中提供了与 Python 中列表操作相对应的函数。例如，列表的 append 方法对应于 PyList_Append()函数，列表的 sort 方法对应于 PyList_Sort()函数，列表的 reverse 方法对应于 PyList_Reverse()函数。其函数原型分别如下。

```
int PyList_Append( PyObject *list, PyObject *item)
int PyList_Sort( PyObject *list)
int PyList_Reverse( PyObject *list)
```

对于 PyList_Append()函数，其参数含义如下。

- list　　要进行操作的列表。
- item　　要参加的项。

对于 PyList_Sort()和 PyList_Reverse()函数，其参数含义相同，具体如下。

- list　　要进行操作的列表。

3. 元组操作

Python/C API 中提供了 PyTuple_New()函数，用以创建一个新的 Python 元组。PyTuple_New()函数返回所创建的元组。其函数原型如下。

```
PyObject* PyTuple_New( Py_ssize_t len)
```

其参数含义如下。

- len　　所创建元组的长度。

当元组创建以后，可以使用 PyTuple_SetItem()函数向元组中添加项。其函数原型如下。

```
int PyTuple_SetItem( PyObject *p, Py_ssize_t pos, PyObject *o)
```

其参数含义如下。

- p　　所进行操作的元组。
- pos　　所添加项的位置索引。
- o　　所添加的项值。

可以使用 Python/C API 中的 PyTuple_GetItem()函数来获取元组中某项的值，其函数原型如下。

```
PyObject* PyTuple_GetItem( PyObject *p, Py_ssize_t pos)
```

其参数含义如下。

- p　　要进行操作的元组。
- pos　　项的位置索引。

当元组创建以后，可以使用_PyTuple_Resize()函数重新调整元组的大小，其函数原型如下。

```
int _PyTuple_Resize( PyObject **p, Py_ssize_t newsize)
```

其参数含义如下。

- p　　指向要进行操作的元组的指针。
- newsize　　新元组的大小。

4. 字典操作

Python/C API 中提供了 PyDict_New()函数，用以创建一个新的字典。PyDict_New()函数返回所创建的字典。其函数原型如下。

```
PyObject* PyDict_New()
```

当字典创建后，可以使用 PyDict_SetItem()函数和 PyDict_SetItemString()函数向字典中添加项。其函数原型分别如下。

```
int PyDict_SetItem( PyObject *p, PyObject *key, PyObject *val)
int PyDict_SetItemString( PyObject *p, const char *key, PyObject *val)
```

其参数含义如下。

- p　　要进行操作的字典。
- key　　添加项的关键字，对于 PyDict_SetItem()函数，其为 PyObject 型；对于 PyDict_SetItemString()函数，其为 char 型。
- val　　添加项的值。

使用 Python/C API 中的 PyDict_GetItem()函数和 PyDict_GetItemString()函数可以获取字典中某项的值。它们都返回项的值。其函数原型分别如下。

```
PyObject* PyDict_GetItem( PyObject *p, PyObject *key)
PyObject* PyDict_GetItemString( PyObject *p, const char *key)
```

其参数含义如下。

- p　　要进行操作的字典。
- key　　添加项的关键字，对于 PyDict_GetItem()函数，其为 PyObject 型；对于 PyDict_GetItemString()函数，其为 char 型。

使用 Python/C API 中的 PyDict_DelItem()函数和 PyDict_DelItemString()函数可以删除字典中的某一项。其函数原型如下。

```
int PyDict_DelItem( PyObject *p, PyObject *key)
int PyDict_DelItemString( PyObject *p, char *key)
```

其参数含义如下。

- p　　要进行操作的字典。
- key　　删除项的关键字，对于 PyDict_DelItem()函数，其为 PyObject 型；对于 PyDict_DelItemString()函数，其为 char 型。

使用 Python/C API 中的 PyDict_Next()函数可以对字典进行遍历。其函数原型如下。

```
int PyDict_Next( PyObject *p, Py_ssize_t *ppos, PyObject **pkey, PyObject **pvalue)
```

其参数含义如下。

- p　　要进行遍历的字典。
- ppos　　字典中项的位置，应该被初始化为 0。
- pkey　　返回字典的关键字。
- pvalue　　返回字典的值。

Python/C API 中提供了与 Python 中字典操作相对应的函数。例如，字典的 item 方法对应于 PyDict_Items()函数。字典的 keys 方法对应于 PyDict_Keys()函数。字典的 values 方法对应于 PyDict_Values()函数。其函数原型分别如下。

```
PyObject* PyDict_Items( PyObject *p)
PyObject* PyDict_Keys( PyObject *p)
PyObject* PyDict_Values( PyObject *p)
```

其参数含义如下。

- p　　要进行操作的字典。

5. 释放资源

在 Python 中，使用“引用计数”机制对内存进行管理，实现垃圾自动回收。在 C/C++中，

使用 Python 对象时，应正确地处理引用计数，否则容易导致内存泄漏。Python/C API 中提供了 Py_CLEAR()、Py_DECREF()等宏来对引用计数进行操作。

当使用 Python/C API 中的函数创建列表、元组、字典等后，就在内存中生成了这些对象的引用计数。在对其完成操作后，应该使用 Py_CLEAR()、Py_DECREF()等宏来销毁这些对象。其原型分别如下。

```
void Py_CLEAR( PyObject *o)
void Py_DECREF( PyObject *o)
```

其参数含义如下。

- o　　要进行操作的对象。

对于 Py_CLEAR()，其参数可以为 NULL 指针，此时，Py_CLEAR()不进行任何操作。而对于 Py_DECREF()，其参数不能为 NULL 指针，否则将导致错误。

6. 模块与函数

使用 Python/C API 中的 PyImport_Import()函数，可以在 C 程序中导入 Python 模块。PyImport_Import()函数返回一个模块对象。其函数原型如下。

```
PyObject* PyImport_Import( PyObject *name)
```

其参数含义如下。

- name　　要导入的模块名。

使用 Python/C API 中的 PyObject_CallObject()函数和 PyObject_CallFunction()函数，可以在 C 程序中调用 Python 中的函数。其参数原型分别如下。

```
PyObject* PyObject_CallObject( PyObject *callable_object, PyObject *args)
PyObject* PyObject_CallFunction( PyObject *callable, char *format, ...)
```

对于 PyObject_CallObject()函数，其参数含义如下。

- callable_object　　要调用的函数对象。
- args　　元组形式的参数列表。

对于 PyObject_CallFunction()函数，其参数含义如下。

- callable_object　　要调用的函数对象。
- format　　指定参数的类型。
- ...　　向函数传递的参数。

使用 Python/C API 中的 PyModule_GetDict()函数可以获得 Python 模块中的函数列表。PyModule_GetDict()函数返回一个字典。字典中的关键字为函数名，值为函数的调用地址。其函数原型如下。

```
PyObject* PyModule_GetDict( PyObject *module)
```

其参数含义如下。

◆ module　　已导入的模块对象。

22.2.3 在 C 中嵌入 Python 实例

在一个名为 pytest.py 的 Python 脚本中，其中包含两个函数，具体代码如下。

```
def sum(l):
    r = 0
    for i in l:
        r = r + i
    print('Using Function sum')
    print('The result is:',)
    print(r)

def strsplit(s, c):
    r = s.split(c)
    return r
```

现在，要在 C++中编写程序，导入上面的 pytest.py 模块，并在 C++中调用模块中的函数进行操作，可按以下方法进行。

在 VS2008 中新建一个名为“EmbPython”的空“Win32 控制台应用程序”项目，并向其添加如下所示的“EmbPython.c”文件。

```
#include <stdio.h>
#include <Python.h>
int main(int argc, char* argv[])
{
    PyObject *modulename, *module, *dic, *func, *args, *rel, *list;
    char *funcname1 = "sum";
    char *funcname2 = "strsplit";
    int i;
    Py_ssize_t s;
    printf("-==在 C 中嵌入 Python==-\n");
    /* Python 解释器的初始化*/
    Py_Initialize();
    if(!Py_IsInitialized())
    {
        printf("初始化失败!");
        return -1;
    }
    /* 导入 Python 模块,并检验是否正确导入 */
    modulename = Py_BuildValue("s", "pytest");
    module = PyImport_Import(modulename);
    if(!module)
    {
        printf("导入 pytest 失败!");
        return  -1;
    }
    /* 获得模块中函数，并检验其有效性 */
    dic = PyModule_GetDict(module);
    if(!dic)
    {
```

```
        printf("错误!\n");
        return -1;
    }
    /* 获得 sum 函数地址并验证 */
    func = PyDict_GetItemString(dic,funcname1);
    if(!PyCallable_Check(func))
    {
        printf("不能找到函数 %s",funcname1);
        return -1;
    }
    /* 构建列表 */
    list = PyList_New(5);
    printf("使用 Python 中的 sum 函数求解下列数之和\n");
    for (i = 0; i < 5; i++)
    {
        printf("%d\t",i);
        PyList_SetItem(list,i,Py_BuildValue("i",i));
    }
    printf("\n");
    /* 构建 sum 函数的参数元组*/
    args = PyTuple_New(1);
    PyTuple_SetItem(args,0,list);
    /* 调用 sum 函数 */
    PyObject_CallObject(func,args);
    /* 获得 strsplit 函数地址并验证*/
    func = PyDict_GetItemString(dic,funcname2);
    if(!PyCallable_Check(func))
    {
        printf("不能找到函数 %s",funcname2);
        return -1;
    }
    /* 构建 strsplit 函数的参数元组 */
    args = PyTuple_New(2);
    printf("使用 Python 中的函数分割以下字符串:\n");
    printf("this is an example\n");
    PyTuple_SetItem(args,0,Py_BuildValue("s","this is an example"));
    PyTuple_SetItem(args,1,Py_BuildValue("s"," "));
    /* 调用 strsplit 函数并获得返回值 */
    rel = PyObject_CallObject(func, args);
    s = PyList_Size(rel);
    printf("结果如下所示:\n");
    for ( i = 0; i < s; i ++)
    {
        printf("%s\n",PyUnicode_AS_UNICODE(PyList_GetItem(rel,i)));
    }
    /* 释放资源 */
    Py_DECREF(list);
    Py_DECREF(args);
    Py_DECREF(module);
    /* 结束 Python 解释器 */
    Py_Finalize();
    printf("按回车键退出程序:\n");
    getchar();
    return 0;
}
```

在 VS2008 中编译运行以上程序，输出如下所示。

```
---在 C 中嵌入 Python---
使用 Python 中的 sum 函数求解下列数之和
0       1       2       3       4
Using Function sum
The result is: 10
使用 Python 中的函数分割以下字符串:
this is an example
结果如下所示:
this
is
an
example
按回车键退出程序:
```

22.3 通过 SWIG 编写 Python 扩展

SWIG 是 Simplified Wrapper and Interface Generator 的简称，SWIG 允许程序员用方便易用的脚本语言公开 C/C++代码。SWIG 支持的语言非常广泛，包括 Ruby、Perl、Tcl 和 Python。因此，使用 SWIG 可以轻松地使用 C/C++为 Python 编写扩展。

22.3.1 在 VS 中使用 SWIG

在使用 SWIG 前，首先需到其官方网站 http://www.swig.org/download.html 下载最新版本，解压下载的压缩包，然后就可以在开发环境中进行设置引用了。

由于在 Windows 平台下，一般使用集成开发环境来创建编译工程，因此为了在集成环境中使用 SWIG，需要对其进行设置。以下步骤是以 VS2008 为例将 SWIG 集成到 VC++中。

step 1 启动 VS2008，单击菜单【工具】|【选项】命令，弹出如图 22-24 所示对话框。

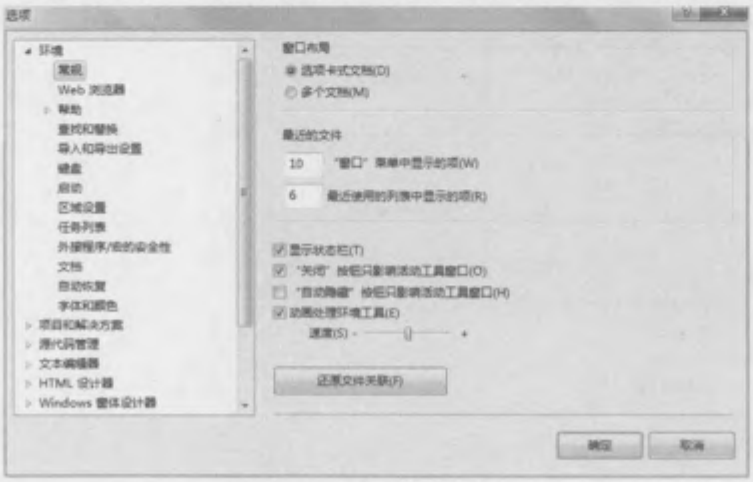


图 22-24 【选项】对话框

step 2 在左侧列表中依次单击选择【项目和解决方案】|【C++目录】项，在右上方选择【显示以下内容的目录】下拉列表框中的【可执行文件】项。接着将 SWIG 所在的目录添加到目录列表中，如图 22-25 所示。

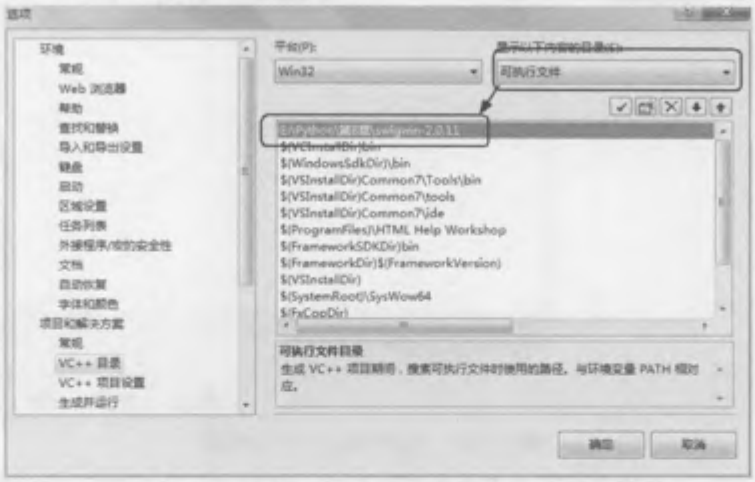


图 22-25 将 SWIG 添加到可执行文件路径列表

step 3 单击【确定】按钮，完成操作。

step 4 在 VS2008 中新建一个名为“useSWIG”的项目，项目类型为“Win32 控制台应用程序”，在向导中设置创建为空的 DLL。向其中添加三个文件，“useSWIG.i”、“useSWIG.c”和“useSWIG_wrap.c”。其中“useSWIG.i”为 SWIG 的接口文件，“useSWIG.c”为编写 Python 扩展的 C 源文件，“useSWIG_wrap.c”为 SWIG 生成的接口文件。

step 5 向“useSWIG.i”中添加以下内容。

```
%module useSWIG
%inline %{
extern void showSWIG();
%}
```

向“useSWIG.c”中添加以下内容。

```
#include <stdio.h>
void showSWIG()
{
    printf("Hi,SWIG!\n");
}
```

step 6 在左侧【解决方案资源管理器】中右击【 useSWIG.i 】文件，从弹出的快捷菜单中选择【属性】命令，弹出【属性页】对话框。在左上方选择【配置】下拉列表框中的【Release】项。选中左侧树形列表中的【自定义生成步骤】，接着在右侧的【命令行】文本框中输入“swig.exe -python "$(InputPath)"，这里的命令是使用 SWIG 对“useSWIG.i”进行编译，生成一个接口文件，生成文件的保存位置在【输出】文本框中设置，这里输入“$(InputName)_wrap.c”（将覆盖最初创建的第 3 个未添加任何内容的文件）。设置界面如图 22-26 所示。

step 7 设置好之后，单击【确定】按钮。



图 22-26 设置自定义生成步骤

step 8 右击【解决方案资源管理器】中的项目名称，从弹出的快捷菜单中选择【属性】命令，打开【属性页】对话框。选中左侧树形列表中的【链接器】|【常规】选项，在右侧将【输出文件】文本框中的内容改为“_useSWIG.pyd”（下画线加上项目名称，扩展名为 pyd），如图 22-27 所示。这是 SWIG 的规定，因为 SWIG 将自动生成一个 Python 脚本文件来调用所编写的 Python 扩展。因此，在使用 Python 扩展的时候应该导入 SWIG 所生成的 Python 脚本，而不是直接导入生成的 Python 扩展文件。



图 22-27 设置输出文件

step 9 单击【确定】按钮，完成项目设置。

step 10 单击菜单【生成】|【批生成】命令，弹出如图 22-28 所示对话框，选择【配置】为“Release”行中的复选项，再单击右上角的【生成】按钮，即可生成 Python 扩展。



图 22-28 【批生成】对话框

step 11 在项目文件所在目录中将生成“_useSWIG.pyd”扩展文件，同时，还将生成一个名为“useSWIG.py”的 Python 脚本文件。创建一个名为“use.py”的 Python 脚本。向其中添加如下所示内容，用来导入“useSWIG.py”模块，并调用上面编译的扩展中的“showSWIG”方法。

```
import useSWIG                  # 此处导入的是 SWIG 生成的 Python 脚本
useSWIG.showSWIG()              # 调用扩展中的 showSWIG 方法
```

运行“useSWIG.py”脚本，将输出如下所示内容。

```
Hi,SWIG!
```

22.3.2 SWIG 接口文件的语法简介

从 22.3.2 节中的例子可以看出，使用 SWIG 编写 Python 扩展非常简单。无须在 C 源文件中使用 Python/C API，只需编写相应的 SWIG 接口文件即可。使用 Python/C API 编写 Python 扩展时需要定义导出函数，并初始化模块名。相应的在 SWIG 接口文件中可以使用%module 设置 Python 扩展的模块名。下面所示的代码定义了一个名为 example 的模块名。

```
%module example
```

在 Python/C API 中，需要使用 PyMethodDef 定义方法列表，而在 SWIG 接口文件中，可以简单地使用%inline 定义方法列表。其形式如下。

```
%inline %{
    包含导出的函数
%}
```

SWIG 将自动生成与 C 中参数类型相对应的 Python 中的类型。按照 22.3.1 节中在 VS2008 中使用 SWIG 的方法将 22.1 节中的“myext”改写为“myswig”。其中，“myswig.c”内容如下。

```
#include <windows.h>
int show(char *message, char *title)
{
```

```
    int r;
    r = MessageBox(NULL,message, title, MB_OK);
    return r;
}
```

其中“myswig.i”内容如下。

```
%module myswig
%inline %{
extern int show(char *message, char *title);
%}
```

编写使用“myswig”的“use.py”脚本，其内容如下。

```
import myswig
myswig.show('Hi,SWIG','MYSWIG')
```

脚本运行后，如图 22-29 所示。



图 22-29　改写为 MYSWIG



如果弹出的对话框中显示乱码，则在 VS2008 中检查一下项目的属性，如图 22-30 所示，在项目属性设置窗口左侧选择【配置属性】|【常规】，然后在右侧将【字符集】设置为【未设置】。



图 22-30　【属性页】对话框

使用 SWIG 编写 Python 扩展可以省去很多代码，而这些代码都将由 SWIG 自动生成。另外，使用 SWIG 不必考虑 C 与 Python 之间数据类型转换的问题。

22.4 Boost.Python 使程序更简单

Boost 是一个可移植的 C++标准库，相当于 STL 的延续和扩充。按照实现的功能，Boost 可分为几十个分类，每个分类针对一个方面提供很多扩展程序库。例如，在字符串和文处理分类中提供了对正则表达式操作的库 Regex。在 Boost 中也提供跨语言混合编程的库，如与 Python 混合编程的 Boost.Python 库，通过这个库，将令使用 C++为 Python 编写扩展变得很简单。

22.4.1 下载编译 Boost.Python

Boost 是以源代码的形式提供的，要使用 Boost.Python 进行编程，首先就需要编译 Boost.Python。由于 Boost 库非常庞大（Boost 1.5.4 下载缩包的大小接近 100MB，解压后接近 400MB），如果没有其他需要，则可以仅对 Boost.Python 进行编译，不用编译其他库。这样可以提高编译的速度（如果将 Boost 全部编译，则需要一小时左右）。下面以 VS2008 为例，介绍编译 Boost.Python 的操作步骤。

step 1 从 Boost 官方网站下载 Boost 库源文件，将其解压至 C 盘根目录（或其他目录）。

step 2 找到安装 VS2008 的目录，在该目录中找到 VC\Bin\vcvars32.bat 文件。

step 3 进入 Windows 7 的命令窗口，执行第 2 步找到的 vcvars32.bat 批处理文件，用来设置 VS2008 的环境参数。操作过程如图 22-31 所示。



图 22-31 设置 VS2008 的环境参数

step 4 在图 22-33 所示命令窗口中输入以下命令，切换到 Boost 的相应目录。

```
cd C:\boost_1_54_0\tools\build\v2
```

step 5 在命令窗口中运行批处理文件 bootstrap.bat，生成 Boost 的编译引擎，操作过程如图 22-32 所示。

step 6 上一步操作将在 C:\boost_1_54_0\tools\build\v2 目录中生成 b2.exe 和 bjam.exe 两个文件，将这两个文件复制到 C:\boost_1_54_0 目录。

step 7 在 C:\boost_1_54_0 目录中创建一个批处理文件 buildpython.bat，其内容如下。

```
@echo 开始编译 python 模块
@pause
```

```
bjam --toolset=msvc-9.0 --with-python link=shared threading=multi variant=release
runtime-link=shared stage
bjam --toolset=msvc-9.0 --with-python link=shared threading=multi variant=debug
runtime-link=shared stage
@echo 编译完成。
@pause
```

在这个批处理文件中，使用 bjam 编译引擎，对 Boost 中的指定模块进行编译，在该命令后面可以有多个参数，这些参数的含义如下。

- --toolset：设置编译器类型。msvc-9.0 是 VS2008 的 C++编译器。
- --with：设置编译哪些模块。-python 表示只编译 python 模块。
- link：设置生成链接库的类型。Shared 为动态链接，static 为静态链接。
- threading：设置链接的线程类型，multi 为多线程模式。
- variant：表示生成调试版（debug）还是发布版（release）。
- runtime：设置运行库链接类型。Shared 为动态链接。
- stage：表示将编译好的库复制到 state 目录下。



图 22-32　生成编译引擎

step 8　接着在命令窗口运行批处理文件 buildpython.bat，开始进行编译。

step 9　经过几分钟的编译后，在 C:\boost_1_54_0\stage\lib 目录中将生成如图 22-33 所示的动态链接库和库文件。

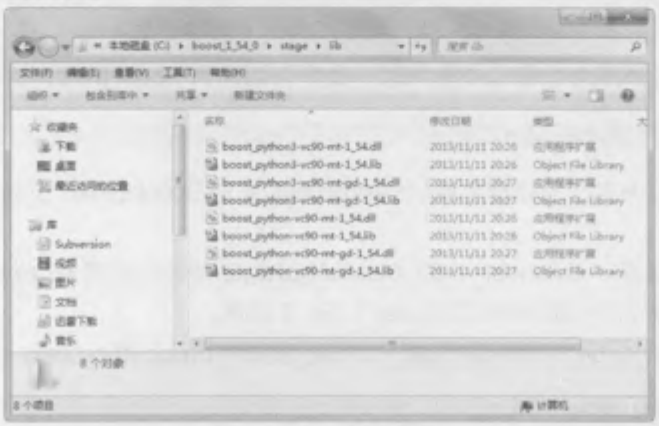


图 22-33　编译结果

step 10 从图 22-33 所示结果可以看出，编译生成的动态链接库和库文件都带有版本号之类的信息，其中文件名中有“gd”的，表示这是调试版（debug），无“gd”的文件则表示是发布版（release），可根据需要使用相应的版本。建议将两个版本的文件都包含，以方便在 VS2008 中编译链接调试版和发布版。

step 11 将 C:\boost_1_54_0\stage\lib 子目录添加到 VS2008 的【库文件】中，如图 22-34 所示。

step 12 类似地，还需要将 C:\boost_1_54_0\目录添加到 VS2008 的【包含文件】中。

完成上述设置后，即可使用 Boost.Python 编写 Python 扩展了。



图 22-34 设置库文件

22.4.2 使用 Boost.Python 简化扩展和嵌入

通过 Boost.Python 可以在 Python 内使用 C++类和函数。和 SWIG 一样，Boost.Python 简化了编写 Python 扩展的代码，而不用使用 Python/C API。与 SWIG 不同的是，Boost.Python 是一个类库，无须再使用接口文件。

1. 初始化和方法列表

在 Boost.Python 中，可以通过使用 BOOST_PYTHON_MODULE 来命名模块名。在 BOOST_PYTHON_MODULE 中，则可以使用 def 来实现使用 Python/C API 定义的方法列表。下面是一个简单的例子。

```
void show()                          // 声明 show 函数
{
    cout << "Boost.Python";
}
BOOST_PYTHON_MODULE(example)         // 使用 BOOST_PYTHON_MODULE 命名模块名为
 "example"
{
    def("show",show);                // 相当于定义方法列表
}
```

2. 导出类

通过 Boost.Python，可以将在 C++中定义的类及其方法属性等导入到 Python 中。以下程序

是在C++中定义一个Message类，然后通过BOOST_PYTHON_MODULE将其导入到Python中。

```
#include <string.h>
#include <boost/python.hpp>
using namespace boost::python;
#pragma comment(lib, "boost_python.lib")
class Message
{
public:
    std::string msg;
    Message(std::string m)
    {
        msg = m;
    }
    void set(std::string m)
    {
        msg = m;
    }
    std::string get()
    {
        return msg;
    }

};
BOOST_PYTHON_MODULE(Message)
{
   class_<Message>("Message", init<std::string>())
        .def("set", &Message::set)
        .def("get", &Message::get)
        ;
}
```

以下代码是在Python中使用编译好的Message模块。

```
>>> import Message
>>> c = Message.Message('hi')
>>> c.get()
'hi'
>>> c.set('Boost.Python')
>>> c.get()
'Boost.Python'
```

3. 类的成员属性

在C++中，类的成员可以使用关键字声明为不同的属性。而在Python中，则依靠类属性的命名方式。使用Boost.Python可以将其C++中类成员的属性传递给Python。下面所示的代码是使用Boost.Python处理类成员的属性。将BOOST_PYTHON_MODULE中的代码改为如下所示。

```
BOOST_PYTHON_MODULE(Message)
{
   class_<Message>("Message", init<std::string>())
      .def("set", &Message::set)
    .def("get", &Message::get)
    .def_readwrite("msg", &Message::msg);
    ;
}
```

此处将Message类中的成员msg设置为可读写。还可以使用“.def_readonly”将其设置为只

读属性。对于类中的私有成员，还可以使用“.add_property”将其操作函数设置为 Python 类中的属性。下面的代码是使用“.add_property”对私有成员进行操作。

```
BOOST_PYTHON_MODULE(Message)
{
class_<Message>("Message",init<std::string>())
    .add_property("msg",&Message::get,&Message::set);
}
```

以下代码是在 Python 中使用编译好的 Message 模块。

```
>>> import Message
>>> s = Message.Message('hi')
>>> s.msg
'hi'
>>> s.msg = 'boot'
>>> s.msg
'boot'
```

4. 类的继承

C++中，类的继承关系也可以通过 Boost.Python 反映到 Python 模块中。下面的代码是将父类和子类分别导入到 Python 模块中。

```
#include <string.h>
#include <boost/python.hpp>
using namespace boost::python;
#pragma comment(lib, "boost_python.lib")
class Message
{
public:
    std::string msg;
    Message(std::string m)
    {
        msg = m;
    }
    void set(std::string m)
    {
        msg = m;
    }
    std::string get()
    {
        return msg;
    }

};
class Msg:public Message
{
public:
    int count;
    Msg(std::string m):Message(m)
    {
    }
    void setcount(int n)
    {
        count = n;
    }
    int getcount()
```

```
    {
        return count;
    }
};
BOOST_PYTHON_MODULE(Message)
{
    class_<Message>("Message",init<std::string>())
        .add_property("msg",&Message::get,&Message::set);
    class_<Msg, bases<Message> >("Msg",init<std::string>())
        .def("setcount", &Msg::setcount)
        .def("getcount", &Msg::getcount);
}
```

5. 运算符重载

在 Python 中，运算符重载实际是类专有方法的重载。在 C++中对运算符重载后，通过 Boost.Python 可以传递给 Python。下面的代码是将 Msg 类的“+”运算符重载，然后通过“.def(self + self)”传递给 Python。

```
class Msg:public Message
{
public:
    int count;
    Msg(std::string m):Message(m)
    {
    }
    void setcount(int n)
    {
        count = n;
    }
    int getcount()
    {
        return count;
    }
    int operator+ (Msg x) const
    {
        int r;
        r = count + x.count;
        return r;
    }
};
BOOST_PYTHON_MODULE(Message)
{
    class_<Message>("Message",init<std::string>())
        .add_property("msg",&Message::get,&Message::set);
    class_<Msg, bases<Message> >("Msg",init<std::string>())
        .def("setcount", &Msg::setcount)
        .def("getcount", &Msg::getcount)
        .def(self + self);
}
```

对于其他的运算符重载也可以使用相同的方法，代码如下所示。

```
.def(self - self)          // 相当于__sub__方法
.def(self * self)          // 相当于__mul__方法
.def(self /self)           // 相当于__div__方法
.def(self < self);         // 相当于__lt__方法
```

6. 使用 Boost.Python 在 C++中嵌入 Python

在 C++中嵌入 Python 的主要问题是 C++与 Python 之间的数据类型的转换。Boost.Python 对 Python/C API 进行了封装，可以使用类似 Python 的方式对数据进行声明、操作等。在 Boost.Python 中，使用 object 类封装了 Python/C API 中的 PyObject*。Boost.Python 定义了与 Python 中列表、字典等对应的类，如表 22-2 所示。

表 22-2 Boost.Python 与 Python 中对应的类

Boost.Python	Python
list	列表
dict	字典
tuple	元组
str	字符串

Boost.Python 提供了 Python 中对象的操作方法。如 str 类具有 lower、split、title、upper 等方法，而 list 类则具有 append、count、extend 等方法。

22.4.3 使用 Pyste 生成代码

Pyste 是 Boost.Python 自带的代码生成器。Pyste 与 SWIG 类似，对于源文件可以按照 C++的形式来写，只要编写相应的接口文件即可生成相应代码。Pyste 需要先安装才能使用。进入 Boost 的安装目录，然后进入“C:\boost_1_54_0\libs\python\pyste\install”目录，运行 python setup.py install，完成 Pyste 安装。

由于 Pyste 需要 GCC-XML 的支持，因此需要到 GCC-XML 的官方网站 http://www.gccxml.org/下载 Windows 版本的 GCC-XML。安装 GCC-XML 后，需要将其安装路径添加到系统 PATH 变量中。另外，Pyste 还需要 ElementTree 的支持，因此需要到其官方网站 http://effbot.org/下载安装。

编写如下所示头文件“Num.h”。

```
class Num
{
    int value;
    void set( int n )
    {
        value = n;
    }
    int get()
    {
        return value;
    }
};
```

编写如下所示接口文件“world.pyste”。

```
Class("Num", "Num.h")
```

由于在 Windows 下有文件路径的问题，因此使用 Pyste 时，最好将其放到“Num.h”和“world.pyste”所在的目录。在 Windows 命令行中进入其目录，运行如下命令。

```
python pyste.py --module=num world.pyste
```

运行命令后将生成“num.cpp”文件，其内容如下。

```
// Boost Includes ==============================================================
#include <boost/python.hpp>
#include <boost/cstdint.hpp>

// Includes
// ============================================================================
#include <Num.h>

// Using
// ============================================================================
using namespace boost::python;

// Module
// ============================================================================
BOOST_PYTHON_MODULE(num)
{
    class_< Num >("Num", init< >())
        .def(init< const Num& >())
    ;
}
```

22.5 本章小结

本章介绍了用 C/C++对 Python 进行扩展和嵌入的方法。首先介绍了用 C/C++扩展 Python 的方法，包括设计 VS2008 开发环境、扩展程序的结构、在扩展程序中使用 MFC。接着介绍了将 Python 嵌入 C/C++程序中的方法。然后介绍了通过 SWIG 简化编写 Python 扩展的方法，最后还介绍了通过 Boost.Python 简化扩展和嵌入的方法。在学习本章内容之前，读者应初步掌握 C/C++程序设计，如果对 C/C++不了解，则可跳过本章。

下一章将进入另一个难点：Python 的多线程编程。

◆ ◆ ◆

第 23 章　多线程编程

本章包括

- 多线程基础
- 线程同步
- 线程间通信
- 微线程——Stackless Python

进程是操作系统中应用程序的执行实例，而线程是进程内部的一个执行单元。当系统创建一个进程后，也就创建了一个主线程。每个进程至少有一个线程，也可以有多个线程。在程序中，使用多线程可以实现并行处理，充分利用 CPU。Python 提供了对多线程的支持，在 Python 中，可以方便地使用多线程进行编程。

23.1　线程基础

在 Python 3 中，通过 threading 模块提供了对多线程编程的支持（在 Python 2.x 中还有一个名为 thread 的模块，threading 模块是对 thread 模块的封装，多数情况下应该使用 threading 模块来进行多线程编程）。下面介绍创建线程的方法。

23.1.1　创建线程

在 Python 中，可以通过继承 threading 类来创建线程。通过继承 threading 模块中 Thread 类来创建新类，在新创建的类中重载 run 方法，然后就可以通过 start 方法创建线程。线程创建后将运行 run 方法。以下代码是使用 threading 模块创建线程。

```
>>> import threading                                    # 导入 threading 模块
>>> class mythread(threading.Thread):                   # 通过继承 Thread 创建类
... def __init__(self,num):                            # 定义初始化方法
...     threading.Thread.__init__(self)                 # 调用父类的初始化方法
...     self.num = num
... def run(self):                                     # 重载 run 方法
...     print('I am ',self.num)
...
>>> t1 = mythread(1)                                   # 生成 mythread 对象
>>> t2 = mythread(2)                                   # 生成 mythread 对象
>>> t3 = mythread(3)                                   # 生成 mythread 对象
>>> t1.start()                                         # 运行线程 t1，实际上是运行 run 方法
I am  1
>>> t2.start()                                         # 运行线程 t2
I am  2
>>> t3.start()                                         # 运行线程 t3
I am  3
```

除了通过继承 threading.Thread 创建类以外，还可以通过使用 threading.Thread 直接在线程中运行函数。以下代码是使用 threading.Thread 直接创建线程。

```
>>> import threading                                    # 导入 threading 模块
>>> def run(x,y):                                       # 定义 run 函数
... for i in range(x,y):
...     print(i)
...
>>> t1 = threading.Thread(target = run,args = (15,20)) # 直接使用 Thread 附加函数，
args 为函数参数
>>> t1.start()                                          # 运行线程
15
16
17
18
19
>>> t2 = threading.Thread(target = run,args = (7,11))  # 直接使用 Thread 附加函数，
args 为函数参数
>>> t2.start()                                          # 运行线程
7
8
9
10
```

23.1.2 Thread 对象中的方法

在 23.1.1 节的例子中，仅使用了 Thread 对象中的 sart 方法，重载了 Thread 对象的 run 方法。当线程被运行时，将运行 run 方法。Thread 对象还有以下几种方法。

1. join 方法

如果一个线程或者函数在执行过程中调用另一个线程，并且必须等待后一线程完成操作后才能继续当前线程的执行，那么在调用线程中可以使用被调用线程的 join 方法。join 方法的原型如下。

```
join([timeout])
```

其参数含义如下。

◆ timeout　　可选参数，线程运行的最长时间。

以下代码是使用 Thread 对象的 join 方法等待线程完成操作。

```
>>> import threading                                  # 导入 threading 模块
>>> import time                                       # 导入 time 模块
>>> class Mythread(threading.Thread):                 # 通过继承 Thread 创建类
... def __init__(self,id):                           # 初始化方法
...     threading.Thread.__init__(self)               # 调用父类的初始化方法
...     self.id = id
... def run(self):                  # 重载 run 方法
...     x = 0
...     time.sleep(60)              # 使用 time 模块中的 sleep 方法让线程休眠 60 秒
...     print(self.id)
```

```
...
>>> def func():                    # 定义函数
... t.start()                      # 运行线程
... for i in range(5):
...     print(i)
...
>>> t = Mythread(2)                # 生成 Mythread 对象
>>> func()                         # 调用函数，运行线程
0                                  # 输出结果中没有线程的输出，func 函数没有等待线程完成
1
2
3
4
>>> def func():                    # 重新定义 func 函数
... t.start()                      # 执行函数，运行线程
... t.join()                       # 调用 join 方法等待线程完成
... for i in range(5):
...     print(i)
...
>>> t = Mythread(3)                # 生成 Mythread 对象
>>> func()                         # 调用函数，运行线程
3                                  # 此为线程输出，然后程序会休眠 60 秒再输出以下内容
0
1
2
3
4
```

2. isAlive 方法

当线程创建后，可以使用 Thread 对象的 isAlive 方法查看线程是否运行。以下代码是使用 Thread 对象的 isAlive 方法查看线程是否运行。

```
>>> import threading                          # 导入 threading 模块
>>> import time                               # 导入 time 模块
>>> class mythread(threading.Thread):         # 通过继承 Thread 创建类
... def __init__(self,id):                    # 初始化方法
...     threading.Thread.__init__(self)       # 调用父类的初始化方法
...     self.id = id
... def run(self):                            # 重载 run 方法
...     time.sleep(5)
...     print(self.id)
...
>>> t = mythread(1)                           # 生成 mythread 对象
>>> def func():                               # 定义函数
... t.start()                                 # 运行线程
... print(t.isAlive())                        # 打印线程状态
...
>>> func()                                    # 调用函数
True                                          # 线程状态
>>> 1                                         # 线程输出
```

3. 线程名

当线程创建后，还可以为线程设置线程名，来区分不同的线程，以便对线程进行控制。线程名可以在类的初始化函数中定义，也可以使用 Thread 对象的 setName 方法设置。使用 Thread 对象的 getName 方法可以获得线程名。以下代码是使用不同方法来设置线程名。

```
>>> import threading                                          # 导入 threading 模块
>>> class mythread(threading.Thread):                         # 通过继承 Thread 创建类
... def __init__(self,threadname):
...     threading.Thread.__init__(self,name = threadname)  # 初始化线程名
... def run(self):                                            # 重载 run 方法
...     print(self.getName())
...
>>> t1 = mythread('t1')                                       # 类实例化，设置线程名
>>> t1.getName()                                              # 调用 getName 方法获得线程名
't1'
>>> t1.setName('T')                                           # 调用 setName 方法设置线程名
>>> t1.getName()                                              # 调用 getName 方法获得线程名
'T'
>>> t2 = mythread('t2')                                       # 类实例化，设置线程名
>>> t2.start()                                                # 运行线程
t2
>>> t2.getName()                                              # 调用 getName 方法获得线程名
't2'
>>> t2.setName('TT')                                          # 调用 setName 方法设置线程名
>>> t2.getName()                                              # 调用 getName 方法获得线程名
'TT'
```

4. daemon 属性

在脚本运行过程中有一个主线程，如果主线程又创建一个子线程，则当主线程退出时，会检验子线程是否完成。如果子线程未完成，则主线程会等待子线程完成后再退出。如果想要主线程退出时，不管子线程是否完成都随主线程退出，则可以通过设置 Thread 对象的 daemon 属性为 True 来达到这种效果。以下代码是给 Thread 对象的 daemon 属性赋值为 True，设置线程随主线程结束而结束。

```
# -*- coding:utf-8 -*-
# file: threaddaemon.py
#
import threading                                  # 导入 threading 模块
import time                                       # 导入 time 模块
class mythread(threading.Thread):                 # 通过继承创建类
    def __init__(self,threadname):               # 初始化方法
        threading.Thread.__init__(self,name = threadname)  # 调用父类的初始化方法
    def run(self):                                # 重载 run 方法
        time.sleep(5)                             # 调用 time.sleep 函数，让线程休眠 5 秒
        print(self.getName())
def func1():                                      # 定义函数 func1
    t1.start()
    print('func1 done')
```

```
def func2():                                    # 定义函数 func2
    t2.start()
    print('func2 done')
t1 = mythread('t1')                             # 类实例化
t2 = mythread('t2')                             # 类实例化
t2.daemon=True                                  # 设置 t2 的 Daemon 标志
func1()                                         # 调用函数 func1
func2()                                         # 调用函数 func2
```

运行 threaddaemon.py 脚本，输出如下所示。

```
func1 done
func2 done
t1
```

由于调用了线程 t2 的 setDaemon 方法，因此当主线程结束时，线程 t2 也随之结束。这里，t2 还没来得及打印自己的线程名，就随主线程一起结束了。将 threaddaemon.py 脚本中的“t2.setDaemon(True)”删除后，保存脚本。重新运行 threaddaemon.py 脚本，输出如下所示。

```
func1 done
func2 done
t1
t2
```

修改后的脚本要等待所有子线程完成后才能退出。因此，线程 t1 和线程 t2 都被执行。如果在交互式模式下运行该实例，则没有区别。因为，在交互式模式下的主线程只有在退出 Python 时才终止。

23.2 线程同步

在多线程环境下，如果多个线程同时对某个数据进行修改，则可能会出现不可预料的结果。为了保证数据被正确修改，就需要对多个线程进行同步。

23.2.1 简单的线程同步

使用 Thread 对象的 Lock 和 RLock 可以实现简单的线程同步。Lock 对象和 RLock 对象都具有 acquire 方法和 release 方法。如果某个数据在某一时刻只允许一个线程进行操作，则可以将操作过程放在 acquire 方法和 release 方法之间。下面的 syn.py 脚本是使用 acquire 方法和 release 方法保持线程同步。

```
# -*- coding:utf-8 -*-
# file: syn.py
#
import threading                                        # 导入 threading 模块
import time                                             # 导入 time 模块
class mythread(threading.Thread):                       # 通过继承创建类
    def __init__(self,threadname):                     # 初始化方法
        threading.Thread.__init__(self,name = threadname)  # 调用父类的初始化方法
```

```
    def run(self):                                    # 重载 run 方法
        global x                                      # 使用 global 表明 x 为全局变量
        lock.acquire()                                # 调用 lock 的 acquire 方法
        for i in range(3):
            x = x + 1
        time.sleep(2)                                 # 调用 sleep 函数，让线程休眠 5 秒
        print(x)
        lock.release()                                # 调用 lock 的 release 方法
lock = threading.RLock()                              # 生成 RLock 对象
tl = []                                               # 定义列表
for i in range(10):
    t = mythread(str(i))                              # 类实例化
    tl.append(t)                                      # 将类对象添加到列表中

x=0                                                   # 将 x 赋值为 0
for i in tl:
    i.start()                                         # 依次运行线程
```

运行脚本后，输出如下。

```
3
6
9
12
15
18
21
24
27
30
```

将 syn.py 脚本中第 11 行的“lock.acquire()”、第 16 行的“lock.release()”和第 17 行的“lock = threading.Lock()”删除后保存脚本。运行修改后的脚本后，输出如下。

```
30
30
30
30
30
30
30
30
30
30
```

修改后的脚本输出都是 30（也就是 i 的最终值）。由于 i 是全局变量，在每个线程对 i 进行操作后又“休眠”了一会。在线程休眠时，Python 解释器已经执行了其他线程，从而使 i 的值不断增加。当所有线程“休眠”结束后，i 的值已被所有线程修改变成了 30，因此最终输出的结果全部为 30。

而在使用 Lock 对象的脚本中，对全局变量 i 的操作是在 acquire 方法和 release 方法之间。Python 解释器每次仅允许一个线程对 i 进行操作。只有当该线程操作完毕，并且结束休眠以后才开始下一个线程，所以使用 Lock 对象的脚本输出是依次递增的。

23.2.2 使用条件变量保持线程同步

Python 的 Condition 对象提供了对复杂线程同步的支持。使用 Condition 对象可以在某些事件触发后才处理数据。Condition 对象除了具有 acquire 方法和 release 方法外，还有 wait 方法、notify 方法、notifyAll 方法等用于条件处理的方法。下面的 P_C.py 脚本是使用 Condition 对象来实现著名的生产者和消费者的关系。

```
# -*- coding:utf-8 -*-
# file: P_C.py
#
import threading                                          # 导入 threading 模块
class Producer(threading.Thread):                         # 定义生产者类
    def __init__(self,threadname):
        threading.Thread.__init__(self,name = threadname)
    def run(self):
        global x
        con.acquire()                                     # 调用 con 的 acquire 方法
        if x == 1000000:
            con.wait()                                    # 调用 con 的 wait 方法
            pass
        else:
            for i in range(1000000):
                x = x + 1
            con.notify()                                  # 调用 con 的 notify 方法
        print x
        con.release()                                     # 调用 con 的 release 方法
class Consumer(threading.Thread):                         # 定义生产者类
    def __init__(self,threadname):
        threading.Thread.__init__(self,name = threadname)
    def run(self):
        global x
        con.acquire()
        if x == 0:
            con.wait()
            pass
        else:
            for i in range(1000000):
                x = x - 1
            con.notify()
        print x
        con.release()
con = threading.Condition()                               # 生成 Condition 对象
x=0
p = Producer('Producer')                                  # 生成生产者对象
c = Consumer('Consumer')                                  # 生成消费者对象
p.start()                                                 # 运行线程
c.start()
p.join()                                                  # 等待线程结束
c.join()
print x
```

运行脚本后，输出如下。

```
1000000
0
0
```

将脚本中第 11、13、18、26、28、33、36 等行中使用 Condition()对象的语句删除。运行修改后的脚本，输出如下。

```
964166
-482930
-482930
```

注意，修改后的脚本每次运行后，输出可能不同。

23.2.3 使用队列让线程同步

Python 中的 Queue 对象也提供了对线程同步的支持。使用 Queue 对象可以实现多生产者和多消费者形成的先进先出的队列。每个生产者将数据依次存入队列，而每个消费者则依次从队列中取出数据。下面的 MP_MC.py 脚本是使用 Queue 对象实现多个生产者和消费者的关系。

```
# -*- coding:utf-8 -*-
# file: MP_MC.py
#
import threading                                    # 导入 threading 模块
import time                                         # 导入 time 模块
import queue                                        # 导入 queue 模块(Python 2.x 中是 Queue)
class Producer(threading.Thread):                   # 定义生产者类
    def __init__(self,threadname):
        threading.Thread.__init__(self,name = threadname)
    def run(self):
        global queue                                # 声明为全局 Queue 对象
        queue.put(self.getName())                   # 调用 put 方法将线程名添加到队列中
        print(self.getName(),'put ',self.getName(),' to queue')
class Consumer(threading.Thread):                   # 定义消费者类
    def __init__(self,threadname):
        threading.Thread.__init__(self,name = threadname)
    def run(self):
        global queue
        print(self.getName(),'get ',queue.get(),'from queue')  # 调用 get 方法获取
队列中内容
queue = queue.Queue()                               # 生成队列对象
plist = []                                          # 生成者对象列表
clist = []                                          # 消费者对象列表
for i in range(10):
    p = Producer('Producer' + str(i))
    plist.append(p)                                 # 添加到生产者对象列表
for i in range(10):
    c = Consumer('Consumer' + str(i))
    clist.append(c)                                 # 添加到消费者对象列表
for i in plist:
    i.start()                                       # 运行生产者线程
    i.join()
for i in clist:
```

```
    i.start()                                    # 运行消费者线程
    i.join()
```

运行脚本后，输出如下。

```
Producer0 put  Producer0  to queue
Producer1 put  Producer1  to queue
Producer2 put  Producer2  to queue
Producer3 put  Producer3  to queue
Producer4 put  Producer4  to queue
Producer5 put  Producer5  to queue
Producer6 put  Producer6  to queue
Producer7 put  Producer7  to queue
Producer8 put  Producer8  to queue
Producer9 put  Producer9  to queue
Consumer0 get  Producer0 from queue
Consumer1 get  Producer1 from queue
Consumer2 get  Producer2 from queue
Consumer3 get  Producer3 from queue
Consumer4 get  Producer4 from queue
Consumer5 get  Producer5 from queue
Consumer6 get  Producer6 from queue
Consumer7 get  Producer7 from queue
Consumer8 get  Producer8 from queue
Consumer9 get  Producer9 from queue
```

23.3 线程间通信

Python 提供了 Event 对象用于线程间的相互通信。实际上，线程同步在一定程度已经实现了线程间的通信。线程同步是每次仅有一个线程对共享数据进行操作，其他线程等待。而 Event 对象是由线程设置信号标志，如果信号标志为真则其他线程等待，直到信号解除。

23.3.1 Event 对象的方法

Event 对象实现了简单的线程通信机制。Event 对象提供了设置信号、清除信号、等待等方法用于实现线程间的通信。

1. 设置信号

使用 Event 对象的 set()方法可以设置 Event 对象内部的信号标志为真。Event 对象提供了 isSet()方法来判断其内部信号标志的状态。当使用 Event 对象的 set()方法后，isSet()方法将返回真。

2. 清除信号

使用 Event 对象的 clear()方法可以清除 Event 对象内部的信号标志，即将其设置为假。当使用 Event 对象的 clear()方法后，isSet()方法将返回假。

3. 等待

Event 对象的 wait 方法只有在其内部信号为真时将很快地执行完成并返回。当 Event 对象的内部信号标志为假时，wait 方法将一直等待，直到其为真时才返回。另外，还可以向 wait 方法传递参数，以设定最长等待时间。

23.3.2 使用 Event 对象实现线程间通信

配合使用 Event 对象的几种方法，可以实现进程间的简单通信。下面的 event.py 脚本即使用 Event 对象实现进程间的通信。

```
# -*- coding:utf-8 -*-
# file: event.py
#
import threading                                    # 导入 threading 模块
class mythread(threading.Thread):
    def __init__(self,threadname):
        threading.Thread.__init__(self,name = threadname)
    def run(self):
        global event                                # 使用全局 Event 对象
        if event.isSet():                           # 判断 Event 对象内部信号标志
            event.clear()                           # 若已设置标志则清除
            event.wait()                            # 调用 wait 方法
            print(self.getName())
        else:
            print(self.getName())
            event.set()                             # 设置 Event 对象内部信号标志
event = threading.Event()                           # 生成 Event 对象
event.set()                                         # 设置 Event 对象内部信号标志
tl = []
for i in range(10):
    t = mythread(str(i))
    tl.append(t)                                    # 生成线程列表

for i in tl:
    i.start()                                       # 运行线程
```

运行脚本后，输出如下。

```
1
0
3
2
5
4
7
6
9
8
```

23.4 微线程——Stackless Python

Stackless Python 是 Python 的一个增强版本。Stackless Python 修改了 Python 的代码，提供了对微线程的支持。微线程是轻量级的线程，与前边所讲的线程相比，微线程在多个线程间切换所需时间更多，占用资源也更少。

23.4.1 Stackless Python 概述

Stackless Python 不是必须的，它只是 Python 的一个修改版本，对多线程编程有更好的支持。如果对线程应用有较高的要求，则可以考虑使用 Stackless Python 来完成。

1. Stackless Python 安装

在安装 Stackless Python 之前，应该先安装 Python，然后根据所安装的 Python 版本到 Stackless Python 的官方网站 http://www.stackless.com/下载相应的版本。对于 Windows 有预编译好的 Stackless Python。以 Python 3.2 为例，下载安装文件 python-3.2.2-stackless.msi（还没有与 Python 3.2.5 对应的安装文件），然后就可以进行安装了。

用鼠标双击安装文件 python-3.2.2-stackless.msi，将启动如图 23-1 所示的【安装】界面。安装过程与本书第 1 章中所介绍的安装 Python 3.2.5 类似。选择安装位置、安装选项等内容之后，就开始复制文件。安装完成后将显示如图 23-2 所示对话框。单击【Finish】按钮即可完成安装。



图 23-1 【安装】界面



图 23-2 【安装完成】对话框

安装完成后，可以在 Python 的交换式环境中输入以下所示代码。

```
import stackless
```

如果没有错误产生，则表示 Stackless Python 已经安装好了；若出现错误，则可能是 Stackless Python 与当前的 Python 版本不兼容，可以考虑使用其他版本的 Python。

2. stackless 模块中的 tasklet 对象

Stackless Python 提供了 stackless 内置模块。stackless 模块中的 tasklet 对象完成了与创建线程类似的功能。使用 tasklet 对象可以像创建线程运行函数那样来运行函数。以下代码是使用 tasklet 对象的部分方法运行函数。

```
>>> import stackless                          # 导入 stackless 模块
>>> def show():                               # 定义 show 函数
...  print('Stackless Python')
...
>>> st = stackless.tasklet(show)()            # 调用 tasklet 添加函数，第 2 个括号为函数参数
>>> st.run()                                  # 调用 run 方法，执行函数
Stackless Python
```

```
>>> st = stackless.tasklet(show)()          # 重新生成 st
>>> st.alive                                # 查看其状态
True
>>> st.kill()                               # 调用 kill 方法结束线程
>>> st.alive                                # 查看其状态
False
>>> stackless.tasklet(show)()               # 直接调用 tasklet
<stackless.tasklet object at 0x011DD3F0>
>>> stackless.tasklet(show)()
<stackless.tasklet object at 0x011DD570>
>>> stackless.run()                         # 调用模块的 run 方法
Stackless Python
Stackless Python
```

3. stackless 模块中的 schedule 对象

stackless 模块中 schedule 对象可以控制任务的执行顺序。当有多个任务时，可以使用 schedule 对象使其依次执行。以下代码是使用 schedule 对象控制任务顺序。

```
>>> import stackless                        # 导入 stackless 模块
>>> def show():                             # 定义 show 函数
... stackless.schedule()                    # 使用 schedule 控制任务顺序
... print(1)
... stackless.schedule()
... print(2)
...
>>> stackless.tasklet(show)()               # 调用 tasklet，生成任务列表
<stackless.tasklet object at 0x011CF830>
>>> stackless.tasklet(show)()
<stackless.tasklet object at 0x011DD570>
>>> stackless.run()                         # 执行任务
1
1
2
2
```

4. stackless 模块中的 channel 对象

使用 stackless 模块中的 channel 对象可以在不同的人之间进行通信，这和线程间的通信类似。使用 channel 对象的 send 方法可以发送数据。使用 channel 对象的 receive 方法可以接收数据。

```
>>> import stackless                        # 导入 stackless 模块
>>> def send():                             # 定义 send 方法
... chn.send('Stackless Python')            # 调用 channel 对象的 send 方法发送数据
... print('I send: Stackless Python')
...
>>> def rec():                              # 定义 rec 方法
... print('I receive:',chn.receive())        # 调用 channel 对象的 receive 方法接收数据
...
>>> stackless.tasklet(send)()               # 调用 tasklet，生成任务列表
<stackless.tasklet object at 0x011DD6B0>
>>> stackless.tasklet(rec)()
<stackless.tasklet object at 0x011DD570>
>>> stackless.run()                         # 执行任务
```

```
I receive: Stackless Python
I send: Stackless Python
```

23.4.2 使用微线程

使用 Stackless Python 的内置模块 stackless 也可以完成多线程编程，使用起来更加方便。下面的 S_P_C.py 脚本是将前面生产者与消费者的代码改写为 Stackless 版，代码更加简洁。

```
# -*- coding:utf-8 -*-
# file: S_P_C.py
#
import stackless                          # 导入 stackless 模块
import queue                              # 导入 queue 模块
def Producer(i):                          # 定义生产者
    global queue                          # 声明为全局 Queue 对象
    queue.put(i)                          # 向队列中添加数据
    print('Producer',i, 'add',i)
def Consumer():                           # 定义消费者
    global queue
    i = queue.get()                       # 从队列中取出数据
    print('Consumer',i, 'get',i)
queue = Queue.Queue()                     # 生成队列对象
for i in range(10):
    stackless.tasklet(Producer)(i)        # 添加生产者任务
for i in range(10):
    stackless.tasklet(Consumer)()         # 添加消费者任务
stackless.run()                           # 执行任务
```

运行脚本后，输出如下。

```
Producer 0 add 0
Producer 1 add 1
Producer 2 add 2
Producer 3 add 3
Producer 4 add 4
Producer 5 add 5
Producer 6 add 6
Producer 7 add 7
Producer 8 add 8
Producer 9 add 9
Consumer 0 get 0
Consumer 1 get 1
Consumer 2 get 2
Consumer 3 get 3
Consumer 4 get 4
Consumer 5 get 5
Consumer 6 get 6
Consumer 7 get 7
Consumer 8 get 8
Consumer 9 get 9
```

23.5 本章小结

本章介绍了 Python 的多线程编程知识，首先介绍了创建线程的方法，以及如何通过 Thread 对象提供的方法控制线程。接着介绍了线程同步的基本操作，介绍了 3 种同步方法：简单同步、通过条件变量同步和通过队列同步。然后介绍了通过 Event 对象实现线程间通信的方法。最后还介绍了支持微线程的 Python 版本，这需要另外下载安装，然后才可以使用微线程。

到本章为止，本书针对 Python 的知识点就介绍完了。下一章将开始编写实际的案例。

◆ ◆ ◆

Part

第3部分 案 例 篇

第 24 章　案例 1：用 Python 优化 Windows

第 25 章　案例 2：用 Python 玩转大数据

第 26 章　案例 3：植物大战僵尸

3

第 24 章　案例 1：用 Python 优化 Windows

本章包括

- 编写脚本创建 GUI
- 扫描垃圾文件
- 删除垃圾文件
- 按名称搜索文件
- 遍历目录
- 使用多线程
- 搜索大文件

由于 Python 的语法简洁、清晰，又具有丰富而强大的类库，因此常被称为“胶水”语言，它能够很轻松地将各种语言模块连接在一起。当然，Python 也能够方便快捷地编写一些常用的工具程序，用其他程序设计语言需要编写很复杂的代码来完成的功能，通过 Python 的类库，可能只需要几行代码就能完成同样的功能。

Python 的这种快速开发特性，在日常计算机维护方面大有用武之地。本章将介绍一些典型的应用场景和实际案例。

24.1　案例概述

Windows 用户都有这样的体会，计算机使用一段时间后，感觉速度越来越慢，不但启动时速度变慢，启动后的运行速度也越来越慢。这是因为，Windows 操作系统在运行过程中会不断地产生垃圾文件，安装、卸载、使用应用程序，也会产生垃圾文件。

这些垃圾文件随着计算机的使用越来越多，如果得不到及时地清理，将会影响到计算机系统的运行速度。

既然知道是由于垃圾文件的原因造成计算机速度变慢，那么解决方法也就有了，就是将这些垃圾文件及时清理掉。

清理垃圾文件的方式有多种。

第一种方法，逐个盘符、目录查看是否存在垃圾文件，若有，则删除。

第二种方法，借助一些工具软件，通过这些软件可以轻松地扫描、删除垃圾文件。

第三种方法，利用所学的 Python 知识，自己编写清理垃圾文件的脚本。

这几种方法各有优缺点。

第一种方法由人工去分辨垃圾文件，可以很自由地决定哪些是垃圾文件，但是费时费力，效率很低。

第二种方法省事省力，但其缺点是只能清理该软件认识的垃圾文件，对于一些由特殊软件产生的垃圾文件则无法辨识。

第三种方法是由 Python 脚本去代替人工辨识垃圾文件，当有特殊的文件类别需要清理时，可以通过修改 Python 脚本快速地删除这些文件。因此，这种方式有第一种方法的灵活性，且效率很高。当然，刚开始编写的清理脚本功能不是很完善，需要用户在使用过程中不断提出新的需求，逐

步完善其功能。

本章的案例主要考虑以下几方面，编写 Python 脚本解决这些问题。

1．扫描垃圾文件，统计垃圾文件数量及大小。

2．删除垃圾文件。

3．扫描计算机中的大文件，并将这些大文件列出来（文件大小可自定义）。

4．查找指定部分文字名称的文件。

本章将编写 Python 脚本来解决这些问题。当然，计算机维护还有很多功能，参照本章编写的 Python 脚本，读者也可以自己编写代码来解决问题。这样，随着需求的不断增多，最终就可以得到一个比较完善的维护计算机的 Python 脚本。

24.2 创建图形化界面

根据 24.1 节提出的需求，本章的案例需要完成多项功能。可以将每一项功能单独编写一个 Python 脚本，在使用时分别执行这些脚本即可。为了方便使用，最好将这些功能集成在一起，并使用图形化界面。这样，仅通过单击鼠标就可选择执行相应的功能了。

24.2.1 编写脚本创建 GUI

在编写创建 GUI 脚本之前，首先应对需求进行分析，划分出功能模块。图 24-1 所示是一种功能模块划分方式，在这个图中，将功能分为“搜索”和“清理”两部分，分别完成搜索文件和清理垃圾文件的功能。另外增加了一个“系统”，主要用来提供退出的功能。

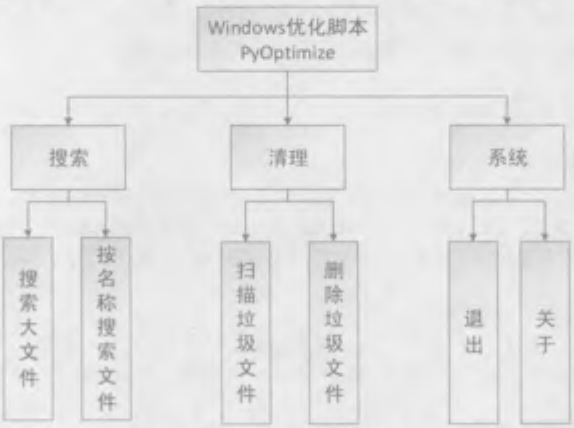


图 24-1 模块划分

从图 24-1 所示的模块划分可以看出，PytOptimize 需要完成 6 项工作，功能不算太多，可以考虑在 GUI 界面中创建 6 个按钮，每个按钮连接一个功能。但是，这种通过按钮的方式不方便以后扩展，如果以后功能扩展到几十个，显然就不适合使用按钮了。对于功能项很多的程序，最好使用菜单。因此，本例也使用菜单来驱动。

编写以下 Python 脚本，通过使用 tkinter 模块创建用户界面。

```
#coding:utf-8
#file: pyOptimize1.py
```

```
import tkinter
import tkinter.messagebox

class Window:
    def __init__(self):
        self.root = tkinter.Tk()

        #创建菜单
        menu = tkinter.Menu(self.root)

        #创建"系统"子菜单
        submenu = tkinter.Menu(menu, tearoff=0)
        submenu.add_command(label="关于...")
        submenu.add_separator()
        submenu.add_command(label="退出")
        menu.add_cascade(label="系统", menu=submenu)

        #创建"清理"子菜单
        submenu = tkinter.Menu(menu, tearoff=0)
        submenu.add_command(label="扫描垃圾文件")
        submenu.add_command(label="删除垃圾文件")
        menu.add_cascade(label="清理", menu=submenu)

        #创建"查找"子菜单
        submenu = tkinter.Menu(menu, tearoff=0)
        submenu.add_command(label="搜索大文件")
        submenu.add_separator()
        submenu.add_command(label="按名称搜索文件")
        menu.add_cascade(label="搜索", menu=submenu)

        self.root.config(menu=menu)

        #创建标签，用于显示状态信息
        self.progress = tkinter.Label(self.root,anchor = tkinter.W,
            text = '状态',bitmap = 'hourglass',compound = 'left')
        self.progress.place(x=10,y=370,width = 480,height = 15)

        #创建文本框，显示文件列表
        self.flist = tkinter.Text(self.root)
        self.flist.place(x=10,y = 10,width = 480,height = 350)

        #为文本框添加垂直滚动条
        self.vscroll = tkinter.Scrollbar(self.flist)
        self.vscroll.pack(side = 'right',fill = 'y')
        self.flist['yscrollcommand'] = self.vscroll.set
        self.vscroll['command'] = self.flist.yview

    def MainLoop(self):
        self.root.title("PyOptimize")
        self.root.minsize(500,400)
        self.root.maxsize(500,400)
        self.root.mainloop()

if __name__ == "__main__" :
```

```
    window = Window()
    window.MainLoop()
```

运行以上脚本，将显示如图 24-2 所示的窗口。在窗口上方显示了 3 个下拉菜单，分别对应图 24-1 所示的模块。在菜单下方是一个列表框，用来显示最终处理结果，列表框右侧添加了一个垂直滚动条，当列表框中的内容太多时，可通过这个滚动条快速移动。在窗口的最下方还添加了一个标签，用来显示当前正在扫描处理的文件。



图 24-2 PyOptimize 窗口

24.2.2 响应菜单事件

在图 24-2 所示窗口中单击选择【系统】|【退出】命令，窗口并不会被关闭。这是因为还没有为菜单创建事件脚本，因此，这里只能通过单击窗口右上角的关闭按钮来关闭窗口。

接下来就为菜单创建事件脚本。将 PyOptimize1.py 另存为 PyOptimize2.py，然后进行修改，将每个子菜单项都添加上 command 设置，修改后的代码如下所示。

```
#coding:utf-8
#file: PyOptimize2.py

import tkinter
import tkinter.messagebox

class Window:
    def __init__(self):
        self.root = tkinter.Tk()

        #创建菜单
        menu = tkinter.Menu(self.root)

        #创建 "系统" 子菜单
        submenu = tkinter.Menu(menu, tearoff=0)
        submenu.add_command(label="关于...",command = self.MenuAbout)
        submenu.add_separator()
        submenu.add_command(label="退出", command = self.MenuExit)
        menu.add_cascade(label="系统", menu=submenu)

        #创建 "清理" 子菜单
```

```
        submenu = tkinter.Menu(menu, tearoff=0)
        submenu.add_command(label="扫描垃圾文件", command = self.MenuScanRubbish)
        submenu.add_command(label="删除垃圾文件", command = self.MenuDelRubbish)
        menu.add_cascade(label="清理", menu=submenu)

        #创建 "查找" 子菜单
        submenu = tkinter.Menu(menu, tearoff=0)
        submenu.add_command(label="搜索大文件", command = self.MenuScanBigFile)
        submenu.add_separator()
        submenu.add_command(label="按名称搜索文件", command = self.MenuSearchFile)
        menu.add_cascade(label="搜索", menu=submenu)

        self.root.config(menu=menu)
        (以下部分省略)
```

在上面的脚本中，为每个子菜单添加了 command（代码中的加粗与斜体部分）。

这时还不能运行修改后的脚本，运行时将显示如下所示的错误提示。

```
Traceback (most recent call last):
  File "PyOptimize2.py", line 58, in <module>
    window = Window()
  File "PyOptimize2.py", line 16, in __init__
    submenu.add_command(label="关于...",command = self.MenuAbout)
AttributeError: 'Window' object has no attribute 'MenuAbout'
```

以上错误提示的含义是 "Window 对象没有 MenuAbout 属性"，是指还没有为 MenuAbout 编写相应的脚本。对于【关于】菜单项，只需要弹出一个对话框，显示当前系统的一些提示信息即可。接着上面的脚本，在 Window 类中编写以下函数。

```
    # "关于" 菜单
    def MenuAbout(self):
        tkinter.messagebox.showinfo("PyOptimize",
            "这是使用 Python 编写的 Windows 优化程序。\n 欢迎使用并提出宝贵意见！ ")
```

编写好 MenuAbout 函数后，再次运行脚本，又会提示没有编写 MenuExit 函数（【退出】菜单项），继续添加以下脚本。

```
    #"退出"菜单
    def MenuExit(self):
        self.root.quit();
```

类似地，还需要对【清理】和【搜索】这两个下拉菜单中的菜单项编写相应的响应函数。由于还没有编写实现其功能的脚本，这里就暂时编写一个替代脚本（例如，当选择该菜单命令时，就显示一个对话框），到后面完成相应的功能脚本时，再替换就行了。具体脚本如下。

```
    #"扫描垃圾文件"菜单
    def MenuScanRubbish(self):
        result = tkinter.messagebox.askquestion("PyOptimize","扫描垃圾文件将需要较
长的时间，是否继续?")
        if result == 'no':
            return
```

```
        tkinter.messagebox.showinfo("PyOptimize","马上开始删除垃圾文件！")

    #"删除垃圾文件"菜单
    def MenuDelRubbish(self):
        result = tkinter.messagebox.askquestion("PyOptimize","删除垃圾文件将需要较
长的时间，是否继续?")
        if result == 'no':
            return
        tkinter.messagebox.showinfo("PyOptimize","马上开始删除垃圾文件！")

    #"搜索大文件"菜单
    def MenuScanBigFile(self):
        result = tkinter.messagebox.askquestion("PyOptimize","扫描大文件将需要较长
的时间，是否继续?")
        if result == 'no':
            return
        tkinter.messagebox.showinfo("PyOptimize","马上开始扫描大文件！")

    #"按名称搜索文件"菜单
    def MenuSearchFile(self):
        result = tkinter.messagebox.askquestion("PyOptimize","按名称搜索文件将需要
较长的时间，是否继续?")
        if result == 'no':
            return
        tkinter.messagebox.showinfo("PyOptimize","马上开始按名称搜索文件！")
```

经过以上修改，脚本又能正常运行了。运行后将显示一个窗口，单击【系统】|【关于】命令，将弹出一个信息提示窗口，如图 24-3 所示。当然，选择其他菜单命令也会弹出相应的信息提示窗口。现在，选择【系统】|【退出】命令，就可以关闭窗口了。



图 24-3 PyOptimize 窗口

24.3 清理垃圾文件

GUI 界面制作完成之后，接下来就该编写脚本来实现相应的功能了。首先来实现清理垃圾文件

的功能，在【清理】下拉菜单中有两个子菜单项：【扫描垃圾文件】和【删除垃圾文件】。要想完成这两项功能，就需要找到垃圾文件所在位置，由于计算机中垃圾文件的分布是随机的，在各子目录中都可能有垃圾文件，因此，首先就需要编写脚本对目录进行遍历。

24.3.1 遍历目录

要遍历目录，有其他程序设计语言经验的读者首先可能会想到，可以使用递归算法来进行遍历。例如，下面的脚本就可以遍历“C:\python32”目录，将该目录及其子目录中的文件名逐个输出。

```
# coding:utf-8
#
# file:traverse.py

import os,os.path

def traverse(pathname):
    for item in os.listdir(pathname):
        fullitem = os.path.join(pathname,item)
        print(fullitem)
        if os.path.isdir(fullitem):                    #判断是否为目录
            traverse(fullitem)

traverse("c:/python32")
```

在上面的脚本中，首先使用 os 模块中的 listdir 方法获取传入参数（目录）中的所有文件和子目录，然后循环处理。具体的处理过程是，首先输出当前项（文件或目录）的名称（使用 join 方法将父目录名和当前项连接起来），然后判断当前项为目录，则递归调用 traverse 函数处理下层子目录。有关递归算法的用法，可参考与算法相关的书籍。

当然，将上述脚本中函数 traverse 的参数换成其他目录，就可以遍历相应目录下的文件了。另外，在上面的遍历脚本中，只是输出文件名，也可以把这里的输出操作替换成其他操作。

在 Python 中，os 模块提供了一个名为 walk 的函数，这个函数可以完成递归的功能。该函数将返回一个元组（root,dirs,files），其中的 root 表示当前目录，dirs 是当前目录下的所有子目录，而 files 则表示当前目录下的所有文件。下面的脚本可以完成对指定目录的遍历，并输出文件名（与 traverse.py 脚本的功能相同）：

```
# coding:utf-8
#
# file:trav_walk.py

import os,os.path

def trav_walk(pathname):
    for root,dirs,files in os.walk(pathname):
        for fil in files:
            fname=os.path.abspath(os.path.join(root,fil))
            print(fname)

trav_walk("c:/python32")
```

在上述脚本中，只对返回元组中的 files 进行了循环输出，对下级子目录的遍历则不用编写脚本，由 walk 函数内部自己处理即可。

24.3.2 扫描垃圾文件

学会编写脚本遍历目录后，再回到案例中来，接下来就可以编写脚本，遍历目录，找出所有的垃圾文件，并进行统计。

在这里，先对垃圾文件进行定义。最简单的方法就是通过文件的扩展名进行界定，如扩展名为tmp、bak、old、wbk、xlk、_mp、log、gid、chk、syd、$$$、@@@和~*等的文件都是垃圾文件（当然这个列表可以扩充）。因此，在遍历目录时就可以判断文件扩展名是否为列表中的内容，若是，则该文件就是垃圾文件。

将上节中编写的 PyOptimize2.py 另存为 PyOptimize3.py，然后进行修改。

首先在脚本开始处编写以下内容，导入 os 和 os.path 模块（处理文件需要使用），然后将定义为垃圾文件的扩展名保存到一个全局变量 rubbishExt 列表中。

```
#coding:utf-8
#file: PyOptimize3.py

import tkinter
import tkinter.messagebox
import os,os.path

rubbishExt=['.tmp','.bak','.old','.wbk','.xlk','._mp','.log','.gid','.chk','.s
yd','.$$$','.@@@','.~*']
```

rubbishExt 列表可进行扩展（注意，这里将 log 定义为垃圾文件，很多日志文件的扩展名为 log，如果调试或分析系统时需要用到，则不要将其定义为垃圾文件进行删除）。

接着，在 Windows 类中编写以下代码，进行垃圾文件扫描。

```
    #扫描垃圾文件
    def ScanRubbish(self):
        global rubbishExt
        total = 0
        filesize = 0
        for root,dirs,files in os.walk("c:/"):
            try:
                for fil in files:
                    filesplit = os.path.splitext(fil)
                    if filesplit[1] == '': #若文件无扩展名
                        continue
                    try:
                        if rubbishExt.index(filesplit[1]) >=0: #扩展名在垃圾文件扩
展名列表中
                            fname = os.path.join(os.path.abspath(root),fil)
                            filesize += os.path.getsize(fname)
                            if total % 20 == 0:
                                self.flist.delete(0.0,tkinter.END)
                            self.flist.insert(tkinter.END,fname + '\n')
                            l = len(fname)
                            if l>60:
                                self.progress['text'] = fname[:30] + '...' +
fname[l-30:l]
                            else:
                                self.progress['text'] = fname
```

```
                        total += 1  #计数
                    except ValueError:
                        pass
                except Exception as e:
                    print(e)
                    pass
        self.progress['text'] = "找到 %s 个垃圾文件，共占用 %.2f M 磁盘空间" %
(total,filesize/1024/1024)
```

最后，修改函数 MenuScanRubbish，只需在原有脚本的下方增加一行，调用 ScanRubbish 函数即可，修改后的 MenuScanRubbish 函数如下。

```
    #"扫描垃圾文件"菜单
    def MenuScanRubbish(self):
        result = tkinter.messagebox.askquestion("PyOptimize","扫描垃圾文件将需要较
长的时间，是否继续?")
        if result == 'no':
            return
        tkinter.messagebox.showinfo("PyOptimize","马上开始扫描垃圾文件！")
        self.ScanRubbish()
```

编写好以上脚本后，运行 PyOptimize3.py 将出现应用程序窗口，选择菜单【清理】|【扫描垃圾文件】命令后会发现一个问题，这时窗口无法响应操作了（如不能移动窗口），经过一段时间之后，标题栏中出现“未响应”提示，如图 24-4 所示。



图 24-4 “未响应”提示

24.3.3 使用多线程

出现这个问题的原因是什么呢?

在 Python 脚本遍历目录时需要用很长的时间，可能是几分钟，也可能是几十分钟甚至几个小时。在执行遍历脚本时，窗口无法响应其他操作。因此，就出现了前面的这些问题。

该怎么办呢？这时就该使用 Python 的多线程操作了，将耗时的操作用另一个后台线程去进行，则前台 GUI 界面仍然可响应用户操作。

在使用多线程时，需要导入 threading 模块，继续修改 PyOptimize.py 脚本，在前面添加导入

threading 模块的脚本，修改内容如下。

```
#coding:utf-8
#file: PyOptimize3.py

import tkinter
import tkinter.messagebox
import os,os.path
import threading
```

然后修改 MenuScanRubbish 函数，添加以创建线程和执行线程的两行脚本，具体如下。

```
    #"扫描垃圾文件"菜单
    def MenuScanRubbish(self):
        result = tkinter.messagebox.askquestion("PyOptimize","扫描垃圾文件将需要较
长的时间，是否继续?")
        if result == 'no':
            return
        tkinter.messagebox.showinfo("PyOptimize","马上开始扫描垃圾文件！")
        #self.ScanRubbish()
        t=threading.Thread(target=self.ScanRubbish)          #创建线程
        t.start()                                            #开始线程
```

再次运行 PyOptimize.py 脚本，在窗口中单击菜单【清理】|【扫描垃圾文件】命令，在窗口界面中即可看到扫描的过程，文本框中将显示垃圾文件名，如图 24-5 左图所示，扫描完成后将在下方状态标签中显示找到的垃圾文件数量和占用的磁盘空间，如右图所示。



图 24-5 扫描垃圾文件

24.3.4 扫描所有磁盘

到目前为止，已经编写完成了扫描垃圾文件的脚本。但是，还有一问题，前面的扫描脚本都是只扫描 C 盘，没有对其他硬盘进行扫描。

接下来编写脚本，对所有磁盘进行扫描。将 PyOptimize3.py 脚本另存为 PyOptimize4.py，然后开始新功能的扩展。

在 Python 的 os 和 os.path 模块中没有提供获取当前计算机盘符的方法（盘符只是 DOS 或

Windows 中才有的概念），不过，也可以通过编写 Python 脚本来获取当前计算机中有哪些盘符。以下脚本就是一种方法。

```
#取得当前计算机的盘符
def GetDrives():
    drives=[]
    for i in range(65,91):
        vol = chr(i) + ':/'
        if os.path.isdir(vol):
            drives.append(vol)
    return tuple(drives)
```

在上述代码中，循环处理整数 65~90，将这些整数通过 Chr 方法转换为字母 A~Z，分别表示 Windows 中的 A 盘至 Z 盘，通过 os.path.isdir 方法判断“C:/”～“Z:/”是不是一个有效的目录，如果是，则表示存在这个盘符。函数最后返回一个元组。

获取所有盘符后，接下来还需要修改 ScanRubbish 函数，将该函数修改为可接受一个元组参数 scanpath（元组参数中保存了盘符或一系列目录），然后在循环扫描代码部分添加一个外层循环，循环处理参数 scanpath 中的每一个元素，具体脚本修改为如下形式（加粗、斜体部分）。

```
    #扫描垃圾文件
    def ScanRubbish(self,scanpath):
        global rubbishExt
        total = 0
        filesize = 0
        for drive in scanpath:
            for root,dirs,files in os.walk(drive):
                try:
                    for fil in files:
                        filesplit = os.path.splitext(fil)
                        if filesplit[1] == '':      #若文件无扩展名
                            continue
                        try:
                            if rubbishExt.index(filesplit[1]) >=0:
                                                    #扩展名在垃圾文件扩展名列表中
                                fname = os.path.join(os.path.abspath(root),fil)
                                filesize += os.path.getsize(fname)
                                if total % 15 == 0:
                                    self.flist.delete(0.0,tkinter.END)

                                l = len(fname)
                                if l > 50:
                                    fname = name[:25] + '...' + fname[l-25:l]
                                self.flist.insert(tkinter.END,fname + '\n')
                                self.progress['text'] = fname
                                total += 1          #计数
                        except ValueError:
                            pass
                except Exception as e:
                    print(e)
                    pass
        self.progress['text'] = "找到 %s 个垃圾文件，共占用 %.2f M 磁盘空间" %
(total,filesize/1024/1024)
```

最后，还需要修改 MenuScanRubbish 函数，在该函数中调用 GetDrives 函数获取所有盘符，

再将这些盘符作为参数传递给 ScanRubbish 函数（在初始化线程时传入），脚本修改为如下形式：

```
    #"扫描垃圾文件"菜单
    def MenuScanRubbish(self):
        result = tkinter.messagebox.askquestion("PyOptimize","扫描垃圾文件将需要较
长的时间，是否继续?")
        if result == 'no':
            return
        tkinter.messagebox.showinfo("PyOptimize","马上开始扫描垃圾文件！")
        #self.ScanRubbish()
        self.drives =GetDrives()
        t=threading.Thread(target=self.ScanRubbish,args=(self.drives,))
        t.start()
```

运行 PyOptimize4.py，单击菜单【清理】|【扫描垃圾文件】命令，Python 将对当前计算机中的所有磁盘进行扫描，最后得到如图 24-6 所示的结果。



图 24-6　扫描垃圾文件

需要注意的是，当计算机中有多个盘符，且保存的文件较多时，这个扫描过程将需较长时间。

24.3.5　删除垃圾文件

接下来编写删除垃圾文件的脚本，将 PyOptimize4.py 另存为 PyOptimize5.py，开始编写新的脚本。

删除垃圾文件的脚本与扫描垃圾文件的脚本相似，只是在找到垃圾文件后增加一条删除文件的脚本代码即可。需要注意的是，在 Windows 中有些临时文件是系统当前正在使用的，无法删除。因此需要增加一个异常处理，当删除过程出现异常时跳过即可。

删除垃圾文件的代码如下：

```
    #删除垃圾文件
    def DeleteRubbish(self,scanpath):
        global rubbishExt
        total = 0
        filesize = 0
        for drive in scanpath:
            for root,dirs,files in os.walk(drive):
```

```
                try:
                    for fil in files:
                        filesplit = os.path.splitext(fil)
                        if filesplit[1] == '': #若文件无扩展名
                            continue
                        try:
                            if rubbishExt.index(filesplit[1]) >=0: #扩展名在垃圾
文件扩展名列表中
                                fname = os.path.join(os.path.abspath(root),fil)
                                filesize += os.path.getsize(fname)

                                try:
                                    os.remove(fname)    #删除文件
                                    l = len(fname)
                                    if l > 50:
                                        fname = fname[:25] + '...' + fname[l-25:l]

                                    if total % 15 == 0:
                                        self.flist.delete(0.0,tkinter.END)

                                    self.flist.insert(tkinter.END,'Deleted '+
fname + '\n')
                                    self.progress['text'] = fname

                                    total += 1  #计数
                                except:                 #不能删除，则跳过
                                    pass
                        except ValueError:
                            pass
                except Exception as e:
                    print(e)
                    pass
        self.progress['text'] = "删除 %s 个垃圾文件，收回 %.2f M 磁盘空间" %
(total,filesize/1024/1024)
```

接着，修改 MenuDelRubbish 函数，在原提示窗口下方添加脚本，创建一个新的线程，在线程构造方法中传入 DeleteRubbish 函数，并给这个函数传入盘符参数，具体修改的内容如下。

```
    #"删除垃圾文件"菜单
    def MenuDelRubbish(self):
        result = tkinter.messagebox.askquestion("PyOptimize","删除垃圾文件将需要较
长的时间，是否继续?")
        if result == 'no':
            return
        tkinter.messagebox.showinfo("PyOptimize","马上开始删除垃圾文件！")
        self.drives =GetDrives()
        t=threading.Thread(target=self.DeleteRubbish,args=(self.drives,))
        t.start()
```

运行 PyOptimize5.py 脚本，即可将当前计算机中各磁盘中的垃圾文件删除。运行的结果如图 24-7 所示。



图 24-7　删除垃圾文件

24.4　搜索文件

其实，编写好扫描垃圾文件的脚本后，再编写搜索文件的脚本就简单得多了。同样是遍历计算机中的所有磁盘，然后再匹配指定条件即可。

24.4.1　搜索大文件

虽然现在计算机中的硬盘容量越来越大，但是随着使用时间的延长，总感觉硬盘空间不够用。每隔一段时间，就会为腾出更多磁盘空间进行一番操作。

其实，经常会有很多非常大的文件隐藏在磁盘的某一个角落，可能这些文件几年都没有使用过。将这些文件找出来，转移到移动硬盘（或直接删除），即可让硬盘腾出一定的空间。

如果人工逐个目录去查找，不仅费时费力，还有可能遗漏。这时，就该 Python 出场了。

承接 24.3 节的案例，继续编写搜索大文件的脚本，将 PyOptimize5.py 脚本另存为 PyOptimize6.py，就可以开始编写脚本了。

在搜索大文件之前，需要用户确定大文件的界限，即多大的文件算大文件。最好的方式就是使用 tkinter 的标准对话框。在前面的脚本中使用了 tkinter.messagebox 模块中的方法来显示信息，而接收用户输入的标准对话框可以使用 tkinter.simpledialog 模块中的相关方法。因此，首先需要导入该模块。下面的脚本是导入 tkinter.simpledialog 模块。

```
#coding:utf-8
#file: PyOptimize6.py

import tkinter
import tkinter.messagebox,tkinter.simpledialog
import os,os.path
import threading
```

接下来修改 MenuScanBigFile 函数，将该函数原有脚本全部删除，改写为以下内容。

```
    #"搜索大文件"菜单
```

```
    def MenuScanBigFile(self):
        s = tkinter.simpledialog.askinteger('PyOptimize','请设置大文件的大小(M)')
        t=threading.Thread(target=self.ScanBigFile,args=(s,))
        t.start()
```

在上面的脚本中，首先调用 tkinter.simpledialog.askinteger 方法，接收用户输入一个文件大小的整数值，接着创建一个线程，在线程构造函数中调用 ScanBigFile 函数搜索大文件，并将用户输入的文件大小 s 作为传入。

最后编写 ScanBigFile 函数，具体内容如下。

```
    #搜索大文件
    def ScanBigFile(self,filesize):
        total = 0
        filesize = filesize * 1024 * 1024
        for drive in GètDrives():
            for root,dirs,files in os.walk(drive):
                for fil in files:
                    try:
                        fname = os.path.abspath(os.path.join(root,fil))
                        fsize = os.path.getsize(fname)

                        self.progress['text'] = fname  #在状态标签中显示每一个遍历的文件
                        if fsize >= filesize:
                            total += 1
                            self.flist.insert(tkinter.END, '%s, [%.2f M]\n' %
(fname,fsize/1024/1024))
                    except:
                        pass
        self.progress['text'] = "找到 %s 个超过 %s M 的大文件" %
(total,filesize/1024/1024)
```

在上述脚本中，遍历计算机中所有磁盘的每一个目录，通过 os.path.getsize 方法获取文件的大小，然后和传入的参数 filesize 进行比较，若比 filesize 大，则找到一个大文件，将其添加到列表框中。

运行 PyOptimize6.py 脚本，单击菜单【搜索】|【搜索大文件】命令，将显示如图 24-8 左图所示的对话框，输入文件大小，单击【OK】按钮开始搜索大文件，最后的结果如右图所示。

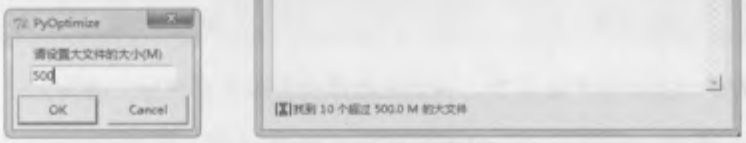


图 24-8　搜索大文件

24.4.2 按名称搜索文件

与搜索大文件类似，要搜索文件名中包含部分字符的文件，也需要用户输入要搜索文件名的部分字符，然后遍历磁盘各目录，逐个比对文件名。

接着 24.4.1 节的案例继续编写按名称搜索文件的脚本，将 PyOptimize7.py 脚本另存为 PyOptimize7.py，就可以开始编写脚本了。

首先修改 MenuSearchFile 函数的脚本，将原来该函数中的内容全部删除，改为以下内容。

```
#"按名称搜索文件"菜单
def MenuSearchFile(self):
    s = tkinter.simpledialog.askstring('PyOptimize','请输入文件名的部分字符')
    t=threading.Thread(target=self.SearchFile,args=(s,))
    t.start()
```

以上脚本中，首先让用户输入文件名中的部分字符，然后创建一个线程，调用 SearchFile 函数，传入用户输入的字符串进行搜索。

接着编写 SearchFile 函数的代码如下。

```
def SearchFile(self,fname):
    total = 0
    fname = fname.upper()
    for drive in GetDrives():
        for root,dirs,files in os.walk(drive):
            for fil in files:
                try:
                    fn = os.path.abspath(os.path.join(root,fil))
                    l = len(fn)
                    if l > 50:
                        self.progress['text'] = fn[:25] + '...' + fn[l-25:l]
                    else:
                        self.progress['text'] = fn

                    if fil.upper().find(fname) >= 0 :
                        total += 1
                        self.flist.insert(tkinter.END, fn + '\n')
                except:
                    pass
    self.progress['text'] = "找到 %s 个文件" % (total)
```

在上述脚本中，使用 upper 函数将传入的文件名部分字符转换为大写，接着遍历计算机磁盘的目录，用 find 方法逐个比较文件名字符，如果有相同的内容（即 find 函数返回值大于等于 0，若 find 函数返回值为-1 则表示没有相同字符串），则表示找到一个文件，将其添加到文本框中即可。

运行 PyOptimize7.py 脚本，单击菜单【搜索】|【按名称搜索文件】命令，将弹出一个对话框，在其中输入要搜索文件名的部分字符，如图 24-9 左图所示，单击【OK】按钮，Python 就开始搜索计算机中是否存在文件名中包含 "PyOptimize" 的文件，经过一段时间的搜索后，最后得到如右图所示的结果。

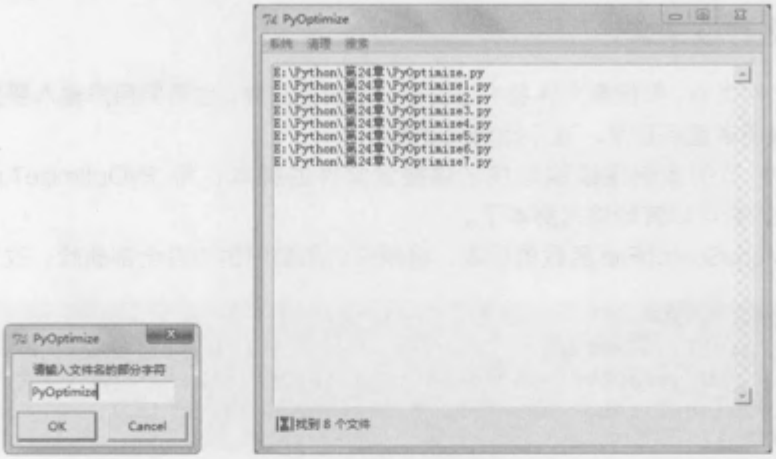


图 24-9 按文件名搜索

24.5 本章小结

本章介绍了用 Python 编写 Windows 优化程序的案例，案例中综合运用了 GUI 设计、多线程、文件目录访问等多个知识点。这个案例主要完成了三部分功能：首先演示了使用 tkinter 模块创建 GUI 界面的全过程，接着介绍了遍历目录脚本的编写方法，在此基础上，通过遍历目录，对计算机中的垃圾文件进行了扫描、删除操作。最后还通过遍历完成文件的搜索功能。在本案例中，为了使图形界面能随时响应用户操作，对遍历目录这种耗时操作使用了多线程技术进行处理。

下一章将通过案例演示 Python 在大数据处理方面的应用。

◆ ◆ ◆

第 25 章　案例 2：用 Python 玩转大数据

本章包括

- 了解大数据处理方式
- 编写 Map 函数处理小文件
- 日志文件的分割
- 编写 Reduce 函数

“大数据（Big Data）”这个术语最早期的引用可追溯到 apache org 的开源项目 Nutch。当时，大数据是用来描述为更新网络搜索索引需要同时进行批量处理或分析的大量数据集。随着谷歌 MapReduce 和 GoogleFile System （GFS）的发布，大数据不再仅用来描述大量的数据，还涵盖了处理数据的速度。

随着云时代的来临，大数据也吸引了越来越多的关注。大数据分析相比于传统的数据仓库应用，具有数据量大、查询分析复杂等特点。

大数据通常用来形容一个公司创造的大量非结构化和半结构化数据，这些数据在下载到关系型数据库用于分析时会花费过多的时间和金钱。大数据分析常和云计算联系在一起，因为实时的大型数据集分析需要像 MapReduce 一样的框架来向数十、数百、甚至数千的电脑分配工作。

在开源领域，Hadoop 的发展如日中天。Hadoop 旨在通过一个高度可扩展的分布式批量处理系统，对大型数据集进行扫描，以产生结果。在 Hadoop 中，可以用两种方式来实现 Map/Reduce。

- Java 的方式，由于 Hadoop 本身是用 Java 来实现的，因此，这种方式最为常见。
- Hadoop Streaming 方式，通过 SHELL/Python/ruby 等各种支持标准输入\输出的语言实现。

由于 Python 在开发效率和高可维护性方法具有很大的优势，因此使用 Python 进行大数据处理也是一种很好的选择。使用 Python 处理大数据，既减少了学习开发语言的难度，又可以较高的开发效率来完成工作。

25.1　案例概述

在 Hadoop 环境下编写 Pyton 脚本，需要预先搭建好 Hadoop 开发环境，比较麻烦。为了演示 Python 在大数据处理方面的应用，本章的案例将不以 Hadoop 环境作为基础，而是以处理某一个或多个大数据量数据为基础，这也符合目前大部分用户的实际应用。根据本章案例，读者可自己编写脚本，处理自己工作中的大数据。

25.1.1　了解大数据处理方式

在 Hadoop 中，采用 MapReduce 编程模型来处理大数据。MapReduce 编程模型用于大规模数据集（大于 1TB）的并行运算。概念“Map（映射）”、“Reduce（规约）”和它们的主要思想，都是从函数式编程语言里借来的，还有从矢量编程语言里借来的特性。这种方式极大地方便了编程

人员在不会分布式并行编程的情况下，将自己的程序运行在分布式系统上。当前的软件实现是指定一个 Map（映射）函数，用来把一组键值对映射成一组新的键值对，指定并发的 Reduce（规约）函数，用来保证所有映射的键值对中的每一个共享相同的键组。

简单地说，在 Hadoop 中通过 MapReduce 编程模型处理大数据时，首先要对大数据进行分割，划分为一定大小的数据，然后将分割的数据分交给 Map 函数进行处理。Map 函数处理后将产生一组规模较小的数据。多个规模较小的数据再提交给 Reduce 函数进行处理，得到一个更小规模的数据或直接结果。

本章的案例将模仿这种 MapReduce 模型又大数据进行处理，下面简单介绍本章需要处理的数据及最终要达到的目标。

25.1.2 处理日志文件

本章案例将处理 Apache 服务器的日志文件 access.log。

Apache 是一个非常流行的 Web（网站）服务器，很多网站都在 Apache 上发布。在网站的日常管理中，经常需要对 apache 网站的日志文件进行分析。通过对这些日志数据进行分析，可以得到很多有用的信息。例如，可分析用户访问量最大的页面，知道用户最关注的商品。还可以分析出用户访问时段，了解网站在一天的哪个时间段访问者最多。还可以从访问者 IP 地址了解访问者的所在区域，了解哪个区域的用户更关注网站……

Apache 服务器的日志文件 access.log 是一个文本格式的文件，可以使用 Windows 的记事本打开。例如，如图 25-1 所示，就是打开该日志文件时所看到的内容。

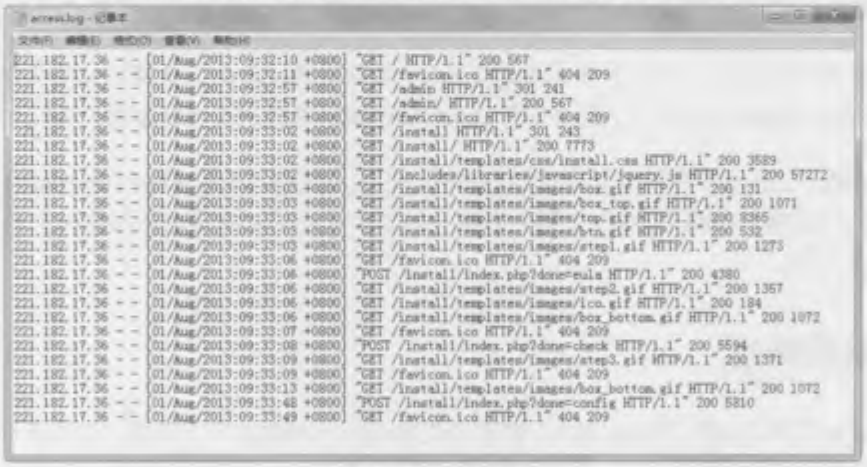


图 25-1 日志文件

从图 25-1 中可以看到，在日志文件中，每一条数据占用一行，每行又分为 7 个部分（用空格隔开），这 7 部分内容依次是：远程主机、空白（E-mail）、空白（登录名）、请求时间、方法+资源+协议、状态代码、发送字节数。

例如，对于下面这一条日志数据：

```
67.198.172.202 - - [01/Dec/2013:18:06:25 +0800] "GET /manager/html HTTP/1.1" 404 210
```

其 7 部分内容分别如下。

- 远程主机：其 IP 地址为 67.198.172.202。
- 空白（E-mail）：这部分为空，在日志中用一个短画线表示。
- 空白（登录名）：这部分为空，在日志中用一个短画线表示。
- 请求时间：为[01/Dec/2013:18:06:25 +0800]。
- 方法+资源+协议：为 GET /manager/html HTTP/1.1。
- 状态代码：为 404。
- 发送字节数：为 210。

如果网站的日志文件比较小，则可直接使用 Windows 的记事本（或其他文本文件编辑器）打开查看。但是，这个日志文件往往很大，很多时候，这个文件大到无法用文本编辑器打开。

其实，提到大数据，可能首先想到的就是上亿条、几十亿条的数据。这在互联网应用中是非常普遍的。例如，若某一个电商网站每天有 20 万访问流量，每位访问者平均打开 10 个页面（每个页面平均产生 8 次请求），则一天将产生 1600 万条访问日志记录数据，一个月就有 48000 万条数据。每条日志数据约为 50~70 个字符，则每个月的日志文件大小约为 25~35GB 之间。

将问题规模缩小一下，即便是访问流量一般的网站，如果每天上千次的流量，则每个月生成的日志文件大小也有几百兆。

对于这么大的文本文件，想打开都很困难，更别说对其进行数据分析了。

25.1.3 案例目标

本章案例将演示用 Python 编写脚本对 apache 日志文件 access.log 进行处理的过程。模似 Hadoop 的 MapReduce 编程模型，按以下流程对数据进行处理。

step 1 首先对大的日志文件进行分割，根据处理计算机的配置，设置一个分割大小的标准，将大的日志文件分割为 *n* 份。

step 2 将分割出来的较小日志文件分别提交给 Map 函数进行处理，这时的 Map 函数可分布在多台计算机中。根据工作量，一个 Map 函数可以处理多个小日志文件。将处理结果保存为一个文本文件，作为 Reduce 函数的输入。

step 3 将各 Map 函数处理的结果提交给 Reduce 函数进行处理，最终得到处理结果。

具体流程如图 25-2 所示。



图 25-2 处理流程



按上述流程编写的 Python 脚本，在测试时可用一个较小的日志文件，最好将日志文件限制在 100MB 以内进行测试，以减少脚本处理的时间，提高开发测试效率。当测试通过之后，再用其处理大的日志。

25.2 日志文件的分割

前面已经提到过，日志文件很大时，是没办法将其直接打开的，这时就可以考虑将其分割为较小的文件。在分割文件时，需要考虑到处理数据的计算机的内存，如果分割的文件仍然较大，则在处理时很容易造成内存溢出。

在 Python 中，对于打开的文件，可以逐行读入数据。因此，分割文件的脚本很简单，具体的脚本如下。

```
#coding:utf-8
#file: FileSplit.py

import os,os.path,time

def FileSplit(sourceFile, targetFolder):
    sFile = open(sourceFile, 'r')
    number = 100000                              #每个小文件中保存 100000 条数据
    dataLine = sFile.readline()
    tempData = []                                #缓存列表
    fileNum = 1
    if not os.path.isdir(targetFolder):          #如果目标目录不存在，则创建
        os.mkdir(targetFolder)
    while dataLine:                              #有数据
        for row in range(number):
            tempData.append(dataLine)            #将一行数据添加到列表中
            dataLine = sFile.readline()
            if not dataLine :                    #没有数据需要保存
                break
        tFilename = os.path.join(targetFolder,os.path.split(sourceFile)[1] +
str(fileNum) + ".txt")
        tFile = open(tFilename, 'a+')            #创建小文件
        tFile.writelines(tempData)               #将列表保存到文件中
        tFile.close()
        tempData = []                            #清空缓存列表
        print(tFilename + " 创建于: " + str(time.ctime()))
        fileNum += 1                             #文件编号

    sFile.close()

if __name__ == "__main__" :
    FileSplit("access.log","access")
```

在上述脚本中，首先设置了每一个分割文件要保存数据的数量，并设置一个空的列表作为缓存，用来保存分割文件的数据。接着打开大的日志文件，逐行读入数据，再将其添加到缓存列表中，当达到分割文件保存数据的数量时，将缓存列表中的数据写入文件。然后，清空缓存列表，继续从大的日志文件中读入数据，重复前面的操作，保存到第 2 个文件中。这样不断重复，最终就可将大

的日志文件分割成小的文件。

在命令行状态中执行 FileSplit.py 脚本，将当前目录中的 access.log 文件分割成小文件，并保存到当前目录的下层 access 目录中。执行结果如图 25-3 所示，从图 25-3 中输出的结果可以看出，在将文件大小为 27MB 的日志文件（约有 25 万条数据）按每个文件 10 万条数据进行分割后，将得到 3 个文件，并且从执行时间来看，在 1 秒钟之内就完成了对 3 个文件的分割、保存操作。



图 25-3 分割文件

再打开 access 目录，可看到分割后得到的小文件，10 万条数据文件的大小约在 11MB 左右，处理这些文件要轻松得多。

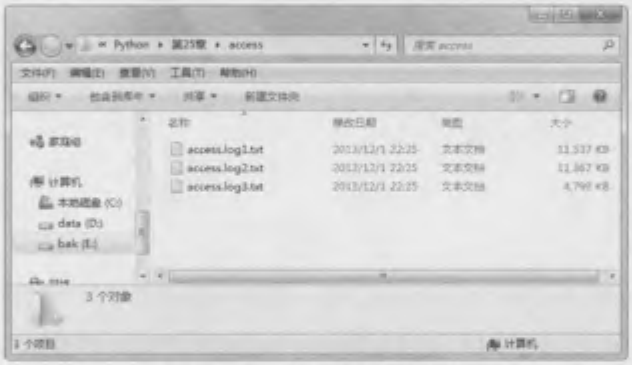


图 25-4 分割得到的小文件

25.3 编写 Map 函数处理小文件

得到分割的小文件之后，接下来就需要编写 Map 函数，对这些小文件进行处理。Map 函数最后得到一个小的数据文件，可能经过处理，将 11MB 大小的文件中的数据进行加工汇总得到一个大小为几百 KB 的文件。再将这个结果文件交给 Reduce 进行处理，这样，就可减轻 Reduce 处理的压力了。

在编写 Map 函数之前，首先需要明确本次处理的目标是什么，即希望从数据中收集哪些信息。目标不同，Map 函数处理的结果将不同。

例如，若需要统计出网站中最受欢迎的页面（即打开次数最多的页面），则在 Map 函数中就需从每条日志中找出页面（日志的第 5 部分，包含“方法+资源+协议”，其中的“资源”就是页面地址），将页面提取出来进行统计。

需要注意的是，在一个 Map 函数中，统计的结果不能作为依据。因此，在这一部分日志文件中，可能 A 页面访问量最大，但在另一部分日志（可能由另一台计算机的 Map 函数处理）中，可能 B 页面的访问量最大。因此，在 Map 函数中，只能是将各页面的访问量分类汇总起来，保存到

一个文件中，交由 Reduce 函数进行最后的汇总。

下面的脚本就可完成分类汇总页面访问量的工作。

```
#coding:utf-8
#file: Map.py

import os,os.path,re

def Map(sourceFile, targetFolder):
    sFile = open(sourceFile, 'r')
    dataLine = sFile.readline()
    tempData = {}                                          #缓存列表
    if not os.path.isdir(targetFolder):                    #如果目标目录不存在，则创建
        os.mkdir(targetFolder)
    while dataLine:                                        #有数据
        p_re = re.compile(r'(GET|POST)\s(.*?)\sHTTP/1.[01]',re.IGNORECASE)
                                                           #用正则表达式解析数据
        match = p_re.findall(dataLine)
        if match:
            visitUrl = match[0][1]
            if visitUrl in tempData:
                tempData[visitUrl] += 1
            else:
                tempData[visitUrl] = 1
        dataLine = sFile.readline()                        #读入下一行数据

    sFile.close()

    tList = []
    for key,value in sorted(tempData.items(),key = lambda k:k[1],reverse = True):
        tList.append(key + " " + str(value) + '\n')

    tFilename = os.path.join(targetFolder,os.path.split(sourceFile)[1] +
"_map.txt")
    tFile = open(tFilename, 'a+')                          #创建小文件
    tFile.writelines(tList)                                #将列表保存到文件中
    tFile.close()


if __name__ == "__main__" :
    Map("access\\access.log1.txt","access")
    Map("access\\access.log2.txt","access")
    Map("access\\access.log3.txt","access")
```

在上面的脚本中，Map 函数打开分割后的小日志文件，然后定义了一个空的字典，用字典来保存不同页面的访问量（用页面链接地址作为字典的键，对应的值就是访问量）。

前面介绍过，日志文件中每一条数据可分为 7 个部分，用空格来隔开。注意，这里最好不用 split 函数对一条日志进行切分，因为日志某些字段内部可能也会出现空格。因此，最好的方式是使用正则表达式来提取页面地址。

得到页面地址后，接着就判断字典中是否已有此地址作为键，若有，则在该键的值上累加 1，表示增加了一次访问。若没有该键，则新建一个键，并设置访问量为 1。

当将（分割后的）小日志文件的每条数据都读入并处理之后，字典 tempData 中就保存了当前

这一部分日志文件中所有页面的访问数据了。最后，对字典进行排序（也可不排序）后生成到一个列表中，再将列表保存到一个后缀为“_map.txt”的文件中，完成当前这一部分日志文件的处理，得到一个较小的结果文件。

执行上述脚本，几秒钟时间就可处理完成，在 access 目录中得到 3 个后缀为“_map.txt”的文件，如图 25-5 所示，从执行结果可以看出，经过 Map 函数的处理，对分割后的约 11MB 大小的文件进行处理后，得到的结果文件大小为 300KB。

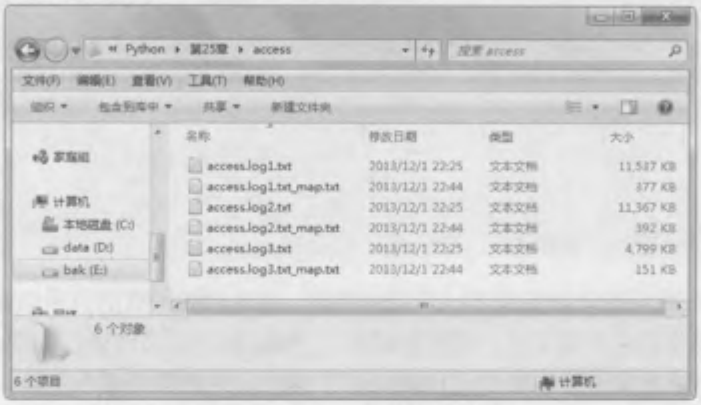


图 25-5 Map 函数执行结果

25.4 编写 Reduce 函数

Reduce 函数进行最后的归集处理。将 Map 函数运算的结果作为 Reduce 函数的输入，经过处理，最后得到一个文件，这个文件就是针对大日志文件的处理结果，而不再是一个部分结果了。

Reduce 函数的处理流程也很简单，就是读入后缀为“_map.txt”的文件，进行数据的归并处理，最后输出一个结果文件。具体的脚本如下。

```
#coding:utf-8
#file: Reduce.py

import os,os.path,re

def Reduce(sourceFolder, targetFile):
    tempData = {}                                          #缓存列表
    p_re = re.compile(r'(.*?)(\d{1,})$',re.IGNORECASE)     #用正则表达式解析数据
    for root,dirs,files in os.walk(sourceFolder):
        for fil in files:
            if fil.endswith('_map.txt'):                   #是 reduce 文件
                sFile = open(os.path.abspath(os.path.join(root,fil)), 'r')
                dataLine = sFile.readline()

                while dataLine:                            #有数据
                    subdata = p_re.findall(dataLine)       #用空格分割数据
                    #print(subdata[0][0]," ",subdata[0][1])
                    if subdata[0][0] in tempData:
                        tempData[subdata[0][0]] += int(subdata[0][1])
                    else:
```

```
                    tempData[subdata[0][0]] = int(subdata[0][1])
                dataLine = sFile.readline()                    #读入下一行数据

            sFile.close()

    tList = []
    for key,value in sorted(tempData.items(),key = lambda k:k[1],reverse = True):
        tList.append(key + " " + str(value) + '\n')


    tFilename = os.path.join(sourceFolder,targetFile + "_reduce.txt")
    tFile = open(tFilename, 'a+')                              #创建小文件
    tFile.writelines(tList)                                    #将列表保存到文件中
    tFile.close()


if __name__ == "__main__" :
    Reduce("access","access")
```

上述脚本中，在循环的外面定义了一个空的字典，用来归并所有的页面访问量数据。接着使用os.walk 函数循环指定目录中的文件，找到后缀为“_map.txt”的文件进行处理。具体处理过程是，逐个将 Map 函数的输出文件（后缀为“_map.txt”）读入，并将数据装入字典。然后对字典进行排序并转换为列表，最后将列表输出到文件，即可得到一个后缀为“_reduce.txt”的文件，在这个文件中保存了日志中所有页面的访问量数据。如果只需要获取访问量前 10（或前 50）的页面，还可以只输出排序后的前 10 条（或前 50 条）数据。

经过文件分割、Map 和 Reduce 处理后，即可将原来大小为 27MB 的文件归集成只有几百 KB 的一个文件，并得到需要的数据。

Reduce 处理得到数据之后，就可以使用 Excel 或其他常用数据处理软件对数据进行分析、输出图表等操作了。当然，也可以在 Python 中继续编写脚本来分析这些数据。

上面的操作是以页面访问量为统计目标进行的数据处理操作。如果有其他目标，则需要编写不同的 Map 和 Reduce 函数来进行处理。例如，若要统计网站每天不同时段的访问量，则在 Map 函数中可使用正则表达式提取日志中的访问时间段，并根据一定的规则进行数据统计。在 Reduce 函数中再根据 Map 函数的输出数据进行归并处理，即可得到所要的数据。

由于 Python 脚本的开发效率较高，因此，开发 Map、Reduce 函数的开发效率也非常高，当统计目标改变后，可以在几分钟就完成对函数的修改，这是其他很多程序设计语言无法做到的。

25.5 本章小结

本章通过编写 Python 脚本模仿了 Hadoop 处理大数据的过程。首先介绍了大数据处理的相关知识，接着编写了 FileSplit 函数对大数据文件进行分割，然后编写 Map 函数处理分割的小文件，并将处理结果保存起来，最后编写 Reduce 函数对这些小文件进行处理，得到最终处理结果。通过本章案例，读者可对大数据处理的流程有一个基本的了解。

下一章将介绍一个有趣的案例：用 Python 编写《植物大战僵尸》游戏。

◆ ◆ ◆

第 26 章　案例 3：植物大战僵尸

本章包括

- 游戏规划设计
- 收集图片素材
- 收集声效素材
- 编写初始脚本
- 编写游戏核心脚本

一般来说，编写游戏程序是一个很复杂的过程，游戏程序需要对图像、声效、控制等进行很好的控制，需要开发者具有较高的程序设计技能。不过，Python 却让游戏开发变得简单了，使用 PyGame 模块，可简单方便地控制游戏的图像、声效等。在本书第 8 章中曾用 PyGame 模块编制了一个简单的游戏程序，在本章，将介绍一个更完整的游戏——植物大战僵尸。

26.1　案例概述

《植物大战僵尸》(Plants vs. Zombies) 是由 PopCap Games 开发的一款益智策略类塔防御战游戏，这款游戏一经发布，就得到众多玩家的喜爱，在 Windows、Mac OS X、iPhone OS 和 Android 系统中都有相应的版本。在这款游戏中，玩家通过植物有效地把僵尸阻挡在入侵的道路上。

26.1.1　游戏效果

由于本书篇幅有限，不可能完整地实现原版《植物大战僵尸》的全部功能，因此，这里将功能进行简化。游戏开始界面如图 26-1 所示。

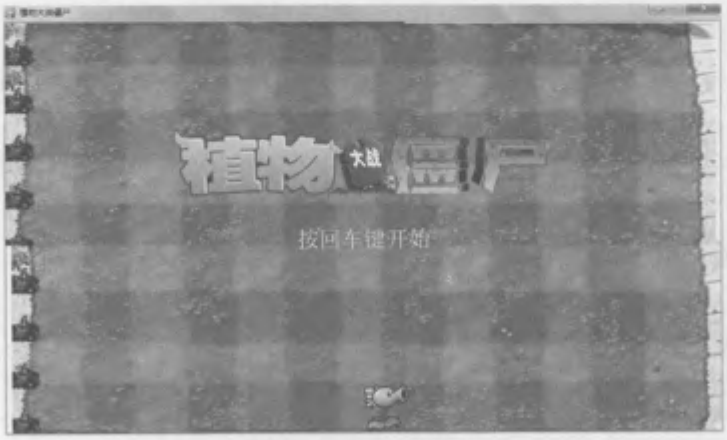


图 26-1　游戏开始界面

游戏进行过程中的界面如图 26-2 所示，左侧是玩家的子弹发射器，右侧是进攻的两种僵尸以及一个攻击僵尸的子弹。

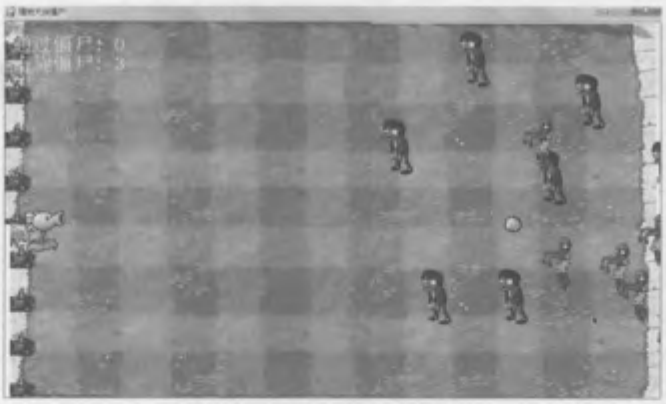


图 26-2　游戏进行过程中的界面

26.1.2 游戏规划设计

在进行游戏设计之前，通常需要对游戏中的目标（即胜负的判定）、角色、角色动作、不同对象的状态转换等进行规划设计。本游戏的初步设计如下。

1. 游戏目标

在本章开发的《植物大战僵尸》游戏中，任务目标仍然是由玩家将入侵的僵尸击毙，以防止其入侵到基地（基地位于屏幕左侧，僵尸从右向左移动进行进攻），如果有 10 个以上僵尸进入基地，则表示基地已被僵尸占领，任务失败。另外，如果玩家的发射器与僵尸接触，则表示发射器被僵尸攻击，任务失败。

2. 游戏角色

- 玩家：玩家只有一个可沿屏幕上下移动的子弹发射器。
- 僵尸：僵尸有两种类型，总数为 50 个。
- 子弹：子弹的数量没有限制，可无限发射。

3. 角色动作

- 玩家的动作：玩家可上下移动发射器（不能左右移动），按空格键或鼠标右键可发射子弹，攻击僵尸。
- 僵尸的动作：僵尸只能沿屏幕从右向左移动，当移动到游戏场地左边沿时自动消失。
- 子弹的动作：子弹只能沿屏幕从左向右移动。

4. 状态的转换

- 玩家：当玩家发射器与僵尸接触时，玩家被击中，游戏结束。
- 僵尸：僵尸的状态转换有以下 3 种。
 - 移动到游戏场地左侧，表示进入基地，僵尸自动消失。
 - 与子弹接触，表示被子弹击中，僵尸消失。
 - 与玩家发射器接触，表示击中玩家，游戏结束。
- 子弹，子弹有两种状态。
 - 子弹向右移动，当移动到游戏场地右边沿时，自动消失。
 - 与僵尸接触，表示子弹击中僵尸，子弹消失。

26.2 收集资源

根据 26.1 节的游戏规划设计，需要有针对性的收集、制作游戏中的素材资源，包括游戏图片、声效等素材。对于商业游戏开发来说，通常需要开发团队自己制作相关的素材。对于本章案例来说，由于这款游戏很普及，作为学习用途，可以直接从原版游戏中借用素材。

下载并在电脑中安装原版的《植物大战僵尸》游戏，从安装目录中就可以找到相关的图片和声效。

26.2.1 收集图片素材

在本章案例中，图片素材用得不多，主要有以下几种。

1. 游戏背景图片

只需要一张游戏背景图片（background.jpg），如图 26-3 所示。由于原版游戏中背景图片大小，不太合适，因此本游戏背景图片通过 Photoshop 进行了重新拼接。

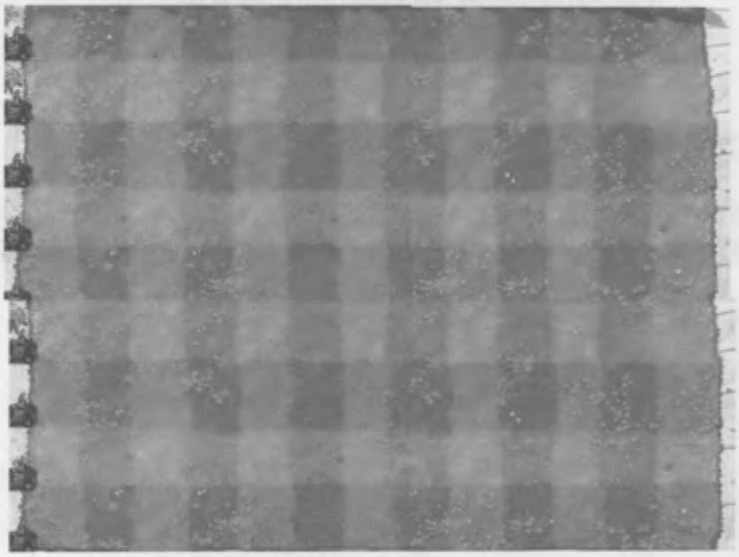


图 26-3 背景图片 background.jpg

2. 僵尸图片

本游戏中有两种僵尸类型，需要有两张不同造型的僵尸图片 Zombie1.png 和 Zombie2.png，如图 26-4 所示。需要注意的是，在游戏中，僵尸需要重叠绘制在背景图片上方，因此，应该使用能保持透明效果的图片格式（如 PNG 或 GIF），不能保存为 JPG 格式。



图 26-4 僵尸图片 Zombie1.png 和 Zombie2.png

3. 玩家发射器图片

玩家发射器图片（Emitter.gif）如图 26-5 所示，也需要使用具有透明效果的 GIF 或 PNG 格式。

4. 子弹图片

子弹图片（Bullet.gif）如图 26-6 所示，也需要使用具有透明效果的 GIF 或 PNG 格式。



图 26-5　发射器图片 Emitter.gif



图 26-6　子弹图片 Bullet.gif

5. 游戏 Logo 图片

游戏 Logo 图片（PvZ_Logo.png）显示在游戏开始界面中，如图 26-7 所示，也需要使用具有透明效果的 GIF 或 PNG 格式。



图 26-7　游戏 Logo 图片 PvZ_Logo.png

26.2.2 收集声效素材

游戏不仅要有好的视觉效果，还应该有一定的声效，才能更好地吸引玩家。本章案例虽然是一个小游戏，但也需要加上一定的声效。如背景音乐、子弹击中僵尸的声效、游戏获胜或失败的声效等。

一般来说，背景音乐需要反复循环播放，可从已有音乐中截取一段能播放 2 分钟左右的片断即可，为了减少音乐文件的大小，可保存为 MP3 格式。而其他声效通常只需要 1~5 秒，所以保存为 WAV 格式即可。PyGame 能方便地播放 MP3 和 WAV 格式的声音。

本章案例使用到的声效文件如下。

- 背景音乐：background.mp3。
- 游戏胜利音效：winmusic.wav。
- 游戏失败音效：losemusic.wav。
- 游戏结束音效：gameover.wav。
- 发射器被僵尸击中音效：hit1.wav。
- 僵尸被子弹击中音效：hit2.wav。

26.3 编写初始脚本

游戏方案设计完成，素材也已收集好之后，就可以开始编写 Python 脚本代码，实现游戏方案中设计的各项内容了。

26.3.1 定义游戏初始环境

根据游戏设计方案，首先编写脚本，定义游戏初始化环境，需要做以下工作。

- 导入脚本中需要用到的各类模块（可在后续脚本编写过程中再增加）。
- 定义场地尺寸。
- 定义角色外形尺寸。
- 定义角色数量。
- 定义角色加入游戏场景的速度。
- 定义角色动作速度。

根据以上分析，编写脚本如下（各变量的作用见右侧的注释）。

```
# coding:utf-8
#
#file PlantsVsZombies.py

import pygame, random, sys, time
from pygame.locals import *

UI_Width = 1024                                  #游戏窗口宽度
UI_Height = 600                                  #游戏窗口高度
FPS = 60                                         #帧频

MAX_Into = 10                                    #最大僵尸通过数(超过此数，游戏结束)
Zombie_Width = 50                                #僵尸宽度
Zombie_Height = 90                               #僵尸高度

Zombie_Total = 50                                #僵尸总数

Zombie_Add_Rate = 30                             #僵尸增加频率

Zombie_Speed = 2                                 #僵尸 1 向左移动的速度
Zombie2_Speed = Zombie_Speed / 2                 #僵尸 2 向左移动的速度

Emitter_Move_Dist = 10                           #发射器每次移动距离
Bullet_Speed = 10                                #子弹的速度
Bullet_Add_Rate = 15                             #子弹列表中添加子弹的频率

Zombies_List = []                                #保存僵尸 1 的列表
Zombies2_List = []                               #保存僵尸 2 的列表
Bullets_List = []                                #子弹列表

Into_Base_Zombies = 0                            #通过的僵尸数量
Score = 0                                        #积分

Add_Zombie_Total = 0                             #添加的僵尸总数

Zombie_Rate_Counter = 0                          #僵尸 1 频率计数器
Zombie2_Rate_Counter = 0                         #僵尸 2 频率计数器
Bullet_Rate_Counter = 40                         #子弹频率计数器

TEXTCOLOR = (255, 255, 255)                      #文本颜色

#上下左右移动标志
To_Left = False
To_Right = False
To_Up = False
To_Down = False
```

```
#射击标志
Shoot = False
```

在上述脚本中，FPS 变量用来设置游戏的最大帧频，即每秒钟游戏最大画面刷新频率，如果将此值改小，则游戏的整体速度都会变慢。而上下左右移动标志可用来设置角色的移动方向，即 To_Left 为 True 时，表示将向左移动，依此类推。而 Shoot 为 True 时，表示处于子弹发射状态。

26.3.2 导入游戏素材

环境设置好之后，接下来就可以通过 PyGame 模块的相关方法来加载游戏中使用的素材，包括图片、声音等。具体代码如下：

```
pygame.init()                                                    #初始化 pygame 模块
mainClock = pygame.time.Clock()
GameWin = pygame.display.set_mode((UI_Width, UI_Height))         #设置游戏场地的大小
pygame.display.set_caption('植物大战僵尸')                        #设置窗口标题
pygame.mouse.set_visible(True)                                   #显示鼠标指针
font = pygame.font.SysFont("simsunnsimsun", 32)                  #设置字体

gameOverSound = pygame.mixer.Sound('gameover.wav')               #游戏结束声音
gameWinSound = pygame.mixer.Sound('winmusic.wav')                #游戏胜利声音
gameLoseSound = pygame.mixer.Sound('losemusic.wav')              #游戏失败声音
gameHit1Sound = pygame.mixer.Sound('hit1.wav')                   #僵尸击中发射器声音
gameHit2Sound = pygame.mixer.Sound('hit2.wav')                   #子弹击中僵尸声音
pygame.mixer.music.load('background.mp3')

EmitterImage= pygame.image.load('Emitter.gif')                   #设置游戏者发射图片
EmitterRect = EmitterImage.get_rect()

bulletImage = pygame.image.load('Bullet.gif')                    # 设置子弹图片
bulletRect = bulletImage.get_rect()

zombieImage = pygame.image.load('Zombie1.png')                   #僵尸 1 图片
Zombie2Image = pygame.image.load('Zombie2.png')                  #僵尸 2 图片
backgroundImage = pygame.image.load('background.jpg')            #设置背景图片
logoImage = pygame.image.load('PvZ_Logo.png')                    #设置 Logo 图片
#将背景图缩放到游戏窗口大小
rescaledBackground = pygame.transform.scale(backgroundImage, (UI_Width, UI_Height))

GameWin.blit(rescaledBackground, (0, 0))                         #把背景画到游戏窗口
GameWin.blit(logoImage,(UI_Width / 2 - 275, 150))
GameWin.blit(EmitterImage, (UI_Width / 2, UI_Height - 70)) #把游戏者的子弹发射器画到屏幕对应位置

pygame.display.update()                                          #刷新屏幕
```

在以上代码中，导入了上一节中收集的图片、声效等素材，然后使用 blit 函数将背景画到游戏窗口中。这里的 blit 是一个很重要的函数，其字面意思是“块传送”（bit block transfer），就是将一个平面的一部分或全部图像整块从这个平面复制到另一个平面。blit 是对窗口重画最多的操作，有两个参数，第一个参数是一个需要传送的对象，第二个参数是左上角的位置坐标。传送完以后必须使用 update 函数进行画面刷新，否则画面是黑的。

在了以上脚本后，就可以运行了。运行后可看到画面一闪而过。这是因为执行完最后的 update 函数后脚本就结束了，为了看清效果，可以在 update 函数下一行中加入等待按钮语句。从图 26-1 所示的游戏开始界面可以看到，提示玩家按回车键进入游戏，否则一直显示开始界面。

对于这样的开始界面，可编写一个死循环，不断检测玩家按钮情况。当检测到回车键时退出死循环。另外，一般游戏通常都提供按 ESC 键退出，这里也应该进行检测。因此，编写一个等待按钮的函数，具体脚本如下。

```
#开始界面时等待游戏者按键
def waitPressKey():
    global
Zombies_List,Zombies2_List,Bullets_List,Into_Base_Zombies,Score,Add_Zombie_Tot
al
    while True:                              #死循环
        for event in pygame.event.get():#获取事件
            if event.type == QUIT:           #事件类型为退出
                quit()                       #退出游戏
            if event.type == KEYDOWN:        #按键事件
                if event.key == K_ESCAPE: #按下 ESC 键
                    quit()                   #退出游戏
                if event.key == K_RETURN: #按下回车
                    init()
                    return                   #退出死循环
```

在以上脚本中，通过 pygame.event.get 函数获取事件，然后判断事件类型，再决定是退出游戏，还是退出当前函数（退出当前函数后，将执行调用该函数的下一条语句，表示程序继续向下运行）。

当前函数是，在检测到退出事件时调用了一个名为 quit 的函数，这个函数将退出游戏，具体脚本如下。

```
# 退出游戏
def quit():
    pygame.quit()
    sys.exit()
```

在 waitPressKey 函数中，在玩家按回车键开始游戏时，将调用 init 函数初始化全局变量，使新开始的游戏为一个新游戏，不会带入上次玩的数据。init 函数的脚本如下。

```
def init():
    global Zombies_List,Zombies2_List,Bullets_List,Into_Base_Zombies,Score,
Add_Zombie_Total
    global To_Left,To_Right,To_Up,To_Down,Shoot
    Zombies_List = []                                    #保存僵尸的列表
    Zombies2_List = []                                   #保存新僵尸的列表
    Bullets_List = []                                    #子弹列表
    Into_Base_Zombies = 0                                #进入基地的僵尸数量
    Score = 0                                            #积分
    Add_Zombie_Total = 0                                 #添加的僵尸总数
    To_Left = To_Right = To_Up = To_Down = Shoot = False #清空各种操作状态
```

注意，这些函数都需要放在调用之前进行定义，否则会出现找不到函数的错误。

将以上函数编写好之后，在 pygame.display.update()的下一行调用 waitPressKey 函数。再次运行脚本，就可以看到如图 26-8 所示的游戏开始界面。这时直接按 ESC 键，画面才会消失。

与图 26-1 对比，图 26-8 所示的开始界面中少了一些提示文字。由于游戏中还有多处要显示文字，因此这里编写一个显示字符串的函数，具体如下。

```
#显示字符串
def DisplayStr(text, font, surface, x, y):
    textobj = font.render(text, 1, TEXTCOLOR)
```

```
    textrect = textobj.get_rect()
    textrect.topleft = (x, y)
    surface.blit(textobj, textrect)
```

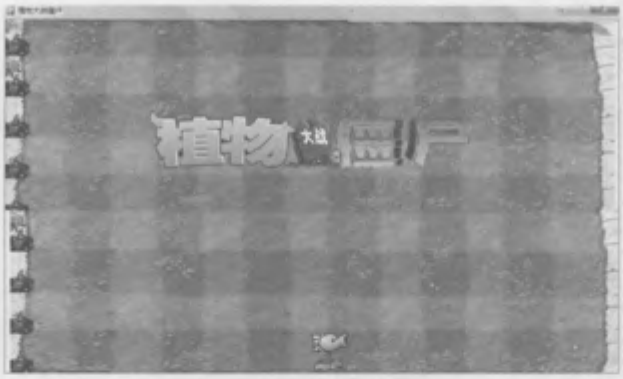


图 26-8　游戏开始界面

在 pygame.display.update()的上一行调用 DisplayStr 函数，以显示提示信息，脚本如下：

```
DisplayStr('按回车键开始', font, GameWin, UI_WIDTH / 2 - 95, 300)
```

这样，游戏开始界面就制作完成了。

26.4　编写游戏核心脚本

前面的脚本只是将游戏开始界面制作出来，接下来开始编写游戏的核心脚本。在这些脚本中实现最初设置的游戏功能。

26.4.1　编写游戏循环脚本

整个游戏脚本可以循环执行，也就是说，当一次游戏结束后（无论胜负），玩家都可以继续玩下一次，直到玩家按 ESC 键退出为止。因此，脚本的主结构可写成以下形式。

```
#进入游戏
while True:
    (游戏)
    waitPressKey()                                    #等待按键
```

上面的脚本很简单，通过 while 死循环来完成重复玩多次的要求，在 waitPressKey 函数中，当玩家按 ESC 键时将调用 quit 函数，结束游戏。

上面的死循环中将是游戏的核心脚本，在这个循环里，需要做以下几项工作。

- 向游戏场景中添加僵尸。
- 向游戏场景中添加子弹。
- 更新各角色的状态。
- 在窗口中绘制出各角色。
- 处理玩家的操作。
- 判断游戏是否结束。

以上每一项处理都需要几行、几十行脚本来完成。若将这些脚本全部集中编写在 while 循环中，脚本不好维护。因此，这里采用函数调用的形式来处理，将这些功能抽取出来编写成多个函数，则主程序中的脚本将比较简单，也方便查看、了解脚本的运行结果。下面的脚本就是 while 循环中的内容，每个函数右侧的注释说明了该函数的作用。

```
#进入游戏
while True:
    EmitterRect.topleft = (30, UI_Height /2) #将发射器放置距左侧 30，垂直位置居中的位置

    pygame.mixer.music.play(-1)              #循环播放音乐

    while True:
        ProcEvent()                          #处理事件
        AddRoles()                           #添加角色到游戏
        RolesStatus()                        #更新角色状态（移动、删除、检查边界）
        ReDraw()                             #重绘画面

        # 检查僵尸是否击中发射器，若击中，则游戏结束
        if EmitterWasHit(EmitterRect, Zombies_List):
            break
        if EmitterWasHit(EmitterRect, Zombies2_List):
            break

        # 检查通过的僵尸数量是否达到最大值，若是，则游戏结束
        if Into_Base_Zombies >= MAX_Into:
            break

        if Score + Into_Base_Zombies >= Zombie_Total:
            break

        mainClock.tick(FPS)                  #设置最大帧率

    # 跳出循环，则停止游戏
    CheckWinOrLose()                         #检查胜负
    pygame.display.update()                  #刷新画面
    waitPressKey()                           #等待按键
```

在上述脚本中，外层 while 循中又嵌套了一层 while 循环。

现在，这个脚本还不能执行，因为还有多个函数没有编写完成，下面逐个介绍这些函数的编写。

26.4.2 处理事件——响应玩家的操作

游戏与玩家的交互是非常重要的操作。在 PyGame 中可通过事件处理的方式来处理玩家的操作，使用 pygame.event.get()函数可获取事件，常见的事件如表 26-1 所示。

表 26-1 pygame 事件

事件	产生途径	参数
QUIT	玩家按下关闭按钮	none
ATIVEEVENT	pygame 被激活或隐藏	gain,state
KEYDOWN	键盘被按下	unicode,key,mod
KEYUP	键盘被放开	key,mod
MOUSEMOTION	鼠标移动	pos,rel,buttons
MOUSEBUTTONDOWN	鼠标按下	pos,button

续表

事件	产生途径	参数
MOUSEBUTTONUP	鼠标放开	pos,button
JOYAXISMOTION	游戏手柄移动	joy,axis,value
JOYBALLMOTION	游戏球移动	joy,axis,value
JOYHATMOTION	游戏手柄移动	joy,axis,value
JOYBUTTONDOWN	游戏手柄按下	joy,button
JOYBUTTONUP	游戏手柄放开	joy,button
VIDEORESIZE	pygame 窗口缩放	size,w,h

下面的脚本可处理键盘中的上、下箭头（或字母 w、s），设置发射器上下移动的状态，并处理键盘的空格键，若是按下状态就设置射击状态为 True，否则设置为 False，类似的，鼠标按下时也设置射击状态为 True。

```
#处理各类事件
def ProcEvent():
    global To_Up,To_Down,Shoot

    for event in pygame.event.get():                    #获取事件
        if event.type == QUIT:                          #退出事件
            quit()

        if event.type == KEYDOWN:                       #键盘按下事件
            if event.key == K_UP or event.key == ord('w'):   #按了向上键或字母 w
                To_Down = False
                To_Up = True
            if event.key == K_DOWN or event.key == ord('s'):     #按了向上键或字母 s
                To_Up = False
                To_Down = True

            if event.key == K_SPACE:                    #按了空格键
                Shoot = True                            #设置发射子弹

        if event.type == KEYUP:                         #键盘放开事件
            if event.key == K_ESCAPE:                   #松开 ESC 键
                quit()

            if event.key == K_UP or event.key == ord('w'):    #松开向上键或字母 w
                To_Up = False
            if event.key == K_DOWN or event.key == ord('s'):
                To_Down = False

            if event.key == K_SPACE:                    #松开空格键
                Shoot = False

        if event.type == MOUSEBUTTONDOWN:               #鼠标按下
            Shoot = True

        if event.type == MOUSEBUTTONUP:                 #鼠标松开
            Shoot = False
```

26.4.3 添加角色到游戏

由于计算机运行速度很快，若每执行一次循环就向游戏场景中将 3 种角色分别添加 1 个，则打开游戏界面时，主界面中就被这些角色填充满了。因此，在脚本最开始的地方设置了僵尸、子弹

的添加频率。如僵尸的添加频率设置为 Zombie_Add_Rate = 30，表示循环执行 30 次才添加 1 次僵尸；类似的，子弹的添加频率设置为 Bullet_Add_Rate = 15，表示循环执行 15 次添加 1 次子弹。

在游戏中，有 3 种角色需要动态添加：两种不同种类的僵尸和子弹。这 3 种角色都通过以下脚本进行添加。

```
#向游戏中增加角色
def AddRoles():
    # 增加僵尸
    global Add_Zombie_Total,Zombie_Rate_Counter,Zombie_Rate_Counter
    global Zombie2_Rate_Counter,Bullet_Rate_Counter,Bullet_Rate_Counter
    if Add_Zombie_Total < Zombie_Total :                    #如果僵尸数量少于总数
        Zombie_Rate_Counter += 1
        if Zombie_Rate_Counter == Zombie_Add_Rate:          #计数器达到，添加僵尸
            Zombie_Rate_Counter = 0                         #计数器清 0
            zombieSize_X = Zombie_Width
            zombieSize_Y = Zombie_Height
            newZombie = {'rect': pygame.Rect(UI_Width, random.randint(10,
UI_Height-zombieSize_Y-10),
                    zombieSize_X, zombieSize_Y),
                    'surface':pygame.transform.scale(zombieImage, (zombieSize_X,
zombieSize_Y)),
                    }

            Zombies_List.append(newZombie)                  #添加到僵尸列表中
            Add_Zombie_Total += 1

        # 增加僵尸 2
        Zombie2_Rate_Counter += 1                           #计数器加 1
        if Zombie2_Rate_Counter == Zombie_Add_Rate:         #若新类僵尸数达到比率
            Zombie2_Rate_Counter = 0                        #计数器清 0

            zombie2Size_X = Zombie_Width
            zombie2Size_Y = Zombie_Height
            newZombie2 = {'rect': pygame.Rect(UI_Width,
random.randint(10,UI_Height-zombie2Size_Y-10),
                    zombie2Size_X, zombie2Size_Y),
                    'surface':pygame.transform.scale(Zombie2Image, (zombie2Size_X,
zombie2Size_Y)),
                }
            Zombies2_List.append(newZombie2)                #添加到僵尸 2 列表中
            Add_Zombie_Total += 1

    # 添加子弹
    Bullet_Rate_Counter += 1
    if Bullet_Rate_Counter >= Bullet_Add_Rate and Shoot == True:
        Bullet_Rate_Counter = 0                             #计数器清 0
        #生成新的子弹
        newBullet = {'rect':pygame.Rect(EmitterRect.centerx+10,
EmitterRect.centery-25,
                bulletRect.width, bulletRect.height),
                'surface':pygame.transform.scale(bulletImage, (bulletRect.width,
bulletRect.height)),
                }
        Bullets_List.append(newBullet)                      #添加到子弹列表
```

26.4.4 更新角色状态

前面的 AddRoles 函数只是将僵尸或子弹这些角色添加到相应的列表中，随着程序的运行（不

断循环），这些角色的状态也在发生变化。例如，当玩家按了键盘中的向上箭头时，子弹发射器应该向上移动，在事件处理函数 ProcEvent 中只是记录了向上的状态（将 To_Up 设置为 True），还没有真正的移动。其他的如子弹、僵尸的状态也需要随着循环的进行而改变。因此，需要编写 RolesStatus 函数来改变这些角色的状态，具体脚本如下。

```
#更新角色状态
def RolesStatus():
    global Score, Into_Base_Zombies
    # 移动子弹发射器
    if To_Up and EmitterRect.top > 30:                          #向上且有向上的空间
        EmitterRect.move_ip(0,-1 * Emitter_Move_Dist)
    if To_Down and EmitterRect.bottom < UI_Height-10:           #向下移动
        EmitterRect.move_ip(0,Emitter_Move_Dist)

    # 移动僵尸 1 （列表中的每个僵尸移动一次）
    for z in Zombies_List:
        z['rect'].move_ip(-1*Zombie_Speed, 0)

    #移动僵尸 2(列表中的每个僵尸移动一次）
    for c in Zombies2_List:
        c['rect'].move_ip(-1*Zombie2_Speed,0)

    # 移动子弹
    for b in Bullets_List:
        b['rect'].move_ip(1 * Bullet_Speed, 0)

    # 删除已移动到左侧的僵尸(遍历判断)
    for z in Zombies_List[:]:
        if z['rect'].left < 0:
            Zombies_List.remove(z)                              #从列表中删除
            Into_Base_Zombies += 1                              #增加进入基地的僵尸数量

    # 删除已移动到左侧的僵尸 2(遍历判断)
    for c in Zombies2_List[:]:
        if c['rect'].left <0:
            Zombies2_List.remove(c)
            Into_Base_Zombies += 1

    #删除已到右边界的子弹
    for b in Bullets_List[:]:
        if b['rect'].right>UI_Width:
            Bullets_List.remove(b)

    # 检查子弹是否击中了僵尸
    for z in Zombies_List:                                      #循环检查每一个僵尸
        if ZombieWasHit(Bullets_List, z):
            Score += 1                                          #积分加 1
            Zombies_List.remove(z)                              #删除僵尸

    for c in Zombies2_List:
        if Zombie2WasHit(Bullets_List, c):
            Score += 1
            Zombies2_List.remove(c)
```

在上面的脚本中，除了移动角色的位置之外，还判断了角色是否已到游戏场景的边界，若是，则从其角色列表中删除。另外，还检查了子弹是否击中僵尸，这里调用了 ZombieWasHit 和 Zombie2WasHit 这两个函数（这两个函数在下面进行编写）来检查。

26.4.5 重绘画面

前面的函数 RolesStatus 只是在角色列表中修改了各角色的状态，接下来，还需要将这些角色绘制到游戏界面中，玩家才能看见。重绘画面的脚本如下。

```
#在场景中重画角色
def ReDraw():
    GameWin.blit(rescaledBackground, (0, 0))

    # 绘制发射器
    GameWin.blit(EmitterImage, EmitterRect)

    # 绘制每一个僵尸
    for z in Zombies_List:
        GameWin.blit(z['surface'], z['rect'])

    for c in Zombies2_List:
        GameWin.blit(c['surface'], c['rect'])

    # 绘制每一枚子弹
    for b in Bullets_List:
        GameWin.blit(b['surface'], b['rect'])

    # 显示得分和通过左边僵尸的数量
    DisplayStr('通过僵尸: %s' % (Into_Base_Zombies), font, GameWin, 10, 20)
    DisplayStr('击毙僵尸: %s' % (Score), font, GameWin, 10, 50)

    pygame.display.update()                                    # 刷新画面
```

26.4.6 判断角色交战状态

在游戏过程中，还需要判断玩家与僵尸的交战状态，如子弹击中了僵尸或僵尸击中了发射器。这些操作，可分别编写函数进行判断，具体脚本如下。

```
#判断僵尸是否击中发射器
def EmitterWasHit(EmitterRect, z):
    for z in Zombies_List:
        if EmitterRect.colliderect(z['rect']):
            gameHit1Sound.play()
            return True
    return False

#判断子弹是否击中了僵尸 1
def ZombieWasHit(Bullets_List, z):
    for b in Bullets_List:
        if b['rect'].colliderect(z['rect']):
            Bullets_List.remove(b) #删除子弹
            gameHit2Sound.play()
            return True
    return False

#判断子弹是否击中了僵尸 2
def Zombie2WasHit(Bullets_List, c):
    for b in Bullets_List:
        if b['rect'].colliderect(c['rect']):
            Bullets_List.remove(b)
            return True
    return False
```

北航 C1724451

26.4.7 判断胜负状态

另外，还需要编写脚本检查游戏的胜负状态。根据规则，当有 10 个以上僵尸进入基地、或者僵尸击中子弹发射器，则玩家失败。当玩家将僵尸击毙（少于 10 个僵尸进入基地），则玩家获胜。根据这个规则编写以下脚本来判断胜负。

```
#检查胜负
def CheckWinOrLose():
    pygame.mixer.music.stop()                                    #停止音乐
    gameOverSound.play()                                         #播放游戏结束音乐
    time.sleep(4)                                                #延时
    gameOverSound.stop()
    GameWin.blit(rescaledBackground, (0, 0))                     #绘制背景
    GameWin.blit(EmitterImage, (UI_Width / 2, UI_Height - 70)) #绘制发射器
    DisplayStr('击毙僵尸: %s' % (Score), font, GameWin, 10, 30)
    DisplayStr('GAME OVER', font, GameWin, UI_Width / 2 - 75, UI_Height / 3)
    DisplayStr('按回车键重玩,或按ESC退出', font, GameWin, UI_Width / 2 - 200, UI_Height
/ 3 + 150)
    if Into_Base_Zombies >= MAX_Into:                            #僵尸通过数量超过最大数
        DisplayStr('僵尸已抢占了你的基地', font, GameWin, UI_Width / 2- 165, UI_Height
/ 3 + 100)
        gameLoseSound.play()
    if EmitterWasHit(EmitterRect, Zombies_List):                 #僵尸击中发射器
        DisplayStr('你被僵尸击中了', font, GameWin, UI_Width / 2 - 110, UI_Height /
3 +100)
        gameLoseSound.play()
    if Score + Into_Base_Zombies >= Zombie_Total:
        DisplayStr('你击败了僵尸的进攻', font, GameWin, UI_Width / 2 - 160, UI_Height
/ 3 +100)
        gameWinSound.play()
```

至此，游戏中的脚本编写完成，接下来就可以进行测试运行了。

执行脚本后将显示如图 26-1 所示界面，按回车键进入游戏界面，如图 26-2 所示。然后按键盘上的上下箭头可上下移动发射器，按空格键（或鼠标键）将发射子弹，当子弹击中僵尸后会发出声音，同时子弹和僵尸全部消失。

26.5 本章小结

游戏开发通常是一个高难度的过程，不过，Python 通过 PyGame 模块可将这些过程简化。本章案例演示了通过 PyGame 模块开发出一个在图像、音效、控制等方面都有不错表现的游戏。本章首先简单介绍了游戏规划设计，接着介绍了收集图片、音效素材的方法，最后调用 PyGame 模块提供的功能编写出游戏脚本。通过本章的案例，读者可了解一般游戏程序的开发流程。

◆ ◆ ◆